THESIS DOCTORAL/DOKTOREGO TESIA

High Performance Scientific Computing in Applications with Direct Finite Element Simulation

Autor/Egilea:
EZHILMATHI KRISHNASAMY

Supervisour/Ikuskatzaile:
JOHAN JANSSON

Bilbao, 2020

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

DOCTORAL THESIS

High Performance Scientific Computing in Applications with Direct Finite Element Simulation

Author:
EZHILMATHI KRISHNASAMY

Supervisour:
JOHAN JANSSON

Bilbao, 2020

This research was carried out at the Basque Center for Applied Mathematics (BCAM) within the CFD Computational Technology (CFDCT) and also at the School of Electrical Engineering and Computer Science(Royal Institue of Technology, Stockholm, Sweden). Which is suported by Fundacion Obra Social “la Caixa“, Severo Ochoa Excellence research centre 2014-2018 SEV-2013-0323, Severo Ochoa Excellence research centre 2018-2022 SEV-2017-0718, BERC program 2014-2017, BERC program 2018-2021, MSO4SC European project, Elkartek. This work has been performed using the computing infrastructure from SNIC (Swedish National Infrastructure for Computing).

Abstract

To predict separated flow including stall of a full aircraft with Computational Fluid Dynamics (CFD) is considered one of the problems of the grand challenges to be solved by 2030, according to NASA [1]. The nonlinear Navier-Stokes equations provide the mathematical formulation for fluid flow in 3-dimensional spaces. However, classical solutions, existence, and uniqueness are still missing. Since brute-force computation is intractable, to perform predictive simulation for a full aircraft, one can use Direct Numerical Simulation (DNS); however, it is prohibitively expensive as it needs to resolve the turbulent scales of order $Re^{\frac{9}{4}}$. Considering other methods such as statistical average Reynolds's Average Navier Stokes (RANS), spatial average Large Eddy Simulation (LES), and hybrid Detached Eddy Simulation (DES), which require less number of degrees of freedom. All of these methods have to be tuned to benchmark problems, and moreover, near the walls, the mesh has to be very fine to resolve boundary layers (which means the computational cost is very expensive). Above all, the results are sensitive to, e.g. explicit parameters in the method, the mesh, etc.

As a resolution to the challenge, here we present the adaptive time-resolved Direct FEM Solution (DFS) methodology with numerical tripping, as a predictive, parameter-free family of methods for turbulent flow. We solved the JAXA Standard Model (JSM) aircraft model at realistic Reynolds number, presented as part of the High Lift Prediction Workshop 3. We predicted lift C_l within 5% error vs. experiment, drag C_d within 10% error and stall 1° within the angle of attack. The workshop identified a likely experimental error of order 10% for the drag results. The simulation is 10 times faster and cheaper when compared to traditional or existing CFD approaches. The efficiency mainly comes from the slip boundary condition that allows coarse meshes near walls, goal-oriented adaptive error control that refines the mesh only where needed and large time steps using a Schur-type fixed-point iteration method, without compromising the accuracy of the simulation results.

As a follow-up, we were invited to the Fifth High Order CFD Workshop, where the approach was validated for a tandem sphere problem (low Reynolds number turbulent flow) wherein a second sphere is placed a certain distance downstream from a first sphere. The results capture the expected slipstream phenomenon, with appx. 2% error. A comparison with the higher-order frameworks Nek500 and PyFR was done. The PyFR framework has demonstrated high effectiveness for GPUs with an unstructured mesh, which is a hard problem in this field. This is achieved by an explicit time-stepping approach. Our study showed that our large time step approach enabled appx. 3 orders of magnitude larger time steps than the explicit time steps in PyFR, which made our method more effective for solving the whole problem.

We also presented a generalization of DFS to variable density and validated against the well-established MARIN benchmark problem. The results show good agreement with experimental results in the form of pressure sensors. Later, we used this methodology to solve two applications in multiphase flow problems. One has to do with a flash rainwater storage tank (Bilbao water consortium), and the second is about designing a nozzle for 3D printing.

In the flash rainwater storage tank, we predicted that the water height in the tank has a significant influence on how the flow behaves downstream of the tank door (valve). For the 3D printing, we developed an efficient design with the focused jet flow to prevent oxidation and heating at the tip of the nozzle during a melting process.

Finally, we presented here the parallelism on multiple GPUs and the embedded system Kalray architecture. Almost all supercomputers today have heterogeneous architectures, such as CPU+GPU or other accelerators, and it is, therefore, essential to develop computational frameworks to take advantage of them.

For multiple GPUs, we developed a stencil computation, applied to geological folds simulation. We explored halo computation and used CUDA streams to optimize computation and communication time. The resulting performance gain was 23% for four GPUs with Fermi architecture, and the corresponding improvement obtained on four Kepler GPUs were 47%.

The Kalray architecture is designed to have low energy consumption. Here we tested the Jacobi method with different communication strategies.

Additionally, visualization is a crucial area when we do scientific simulations. We developed an automated visualization framework, where we could see that task parallelization is more than 10 times faster than data parallelization. We have also used our DFS in the cloud computing setting to validate the simulation against the local cluster simulation. Finally, we recommend the easy pre-processing tool to support DFS simulation.

Acknowledgement

I would like to thank my supervisor Johan Jansson for his guidance throughout my thesis. And also would like to thank my co-supervisors Jose Antonio Lozano and Arghir Dani Zarnescu. A special thanks go to Xing Cai, Johan Hoffman, and Leon Kos for their valuable advice on a critical situation.

I would also like to thank the La Caixa foundation for my PhD scholarship, which has helped me to stay 3 years at KTH, Sweden. On the other hand, BCAM is very supportive and flexible, including directors and administrative staff. I am really thankful for that. I would like to thank all my friends, co-authors, and colleagues from BCAM and KTH.

I wish to thank all the industrial collaborators from Bilbao and Sweden, which is also helped me to gain more knowledge in the applied field. Finally, I would like to thank my mom and sister for their love and support, as always.

Ezhilmathi Krishnasamy
San Sebastian,
Basque Country, 2020

1. Resumen

Esta tesis describe una dinámica de fluidos computacional (CFD), que resuelve los problemas de mecánica de fluidos utilizando los métodos numéricos y computacionales; en concreto, muestra modelos matemáticos modernos eficientes y su simulación por computadora. Las ecuaciones que rigen el flujo de fluido se describen mediante el impulso de continuidad (masa) (segunda ley de Newton) y la ecuación de energía (primera ley de la termodinámica). Antes de entrar en detalles sobre esta tesis, nos gustaría ofrecer una visión general del CFD genérico y su enfoque de arriba a abajo de manera genérica. CFD resuelve las ecuaciones diferenciales parciales por medio de un sistema de ecuaciones usando las computadoras. Mediante este método podríamos resolver toda la gama de problemas (problemas del mundo real) que surgen en nuestra vida cotidiana, incluyendo los siguientes:

- Aerodinámica de aviones y automóviles (elevación y arrastre).
- Ingeniería marina para apoyar la estructura off-shore.
- Diseño de turbinas eólicas (diseño de palas para producir la máxima energía de la turbina).
- Control de la contaminación del aire en ingeniería ambiental.
- Predicción del tiempo en meteorología.
- Ingeniería biomédica en la simulación cardíaca de los flujos sanguíneos a través de arterias y venas.
- Hidrodinámica del barco (por ejemplo, submarino).
- Motores a reacción (diseño de turbomaquinaria).
- Ahora viendo esos ejemplos, uno puede llegar a la conclusión de que CFD es esencial e importante para la sociedad ecológica.

Solo en las últimas décadas, el CFD se ha utilizado ampliamente en ciencia e ingeniería. Por ejemplo, desde la década de 1960 en adelante, CFD se ha utilizado en las industrias aeroespaciales para diseño e I + D. Anteriormente su uso no estaba extendido, debido principalmente a la falta de disponibilidad de recursos computationales.

Es natural preguntarse lo siguiente: ¿cuál es el beneficio de usar el CFD en lugar de los métodos tradicionales? Los métodos tradicionales han existido durante algún tiempo con la disponibilidad de ecuaciones diferenciales y análisis numéricos. Por contra, el CFD presenta la siguiente serie de ventajas:

- Es más barato debido a que require menos mano de obra y tiempo.
- Es más rápido al usar recursos computacionales.
- El trabajo se puede hacer en paralelo.
- El resultado de los resultados se puede utilizar para otros fines.
- Todo el proceso es cuantitativo predictivo.
- Se pueden analizar sistemas complejos (por ejemplo, estudios de seguridad y escenarios de accidentes).

Hasta ahora, hemos visto la importancia de CFD y sus aplicaciones. Ahora le brindaremos detalles genéricos sobre el flujo de trabajo del CFD. El proceso de CFD se puede separar en tres áreas; son:

- pre-procesamiento
- Solver
- Post-procesamiento

1.1. Pre-procesamiento

Se trata de convertir el modelo real en el modelo de computadora para la simulación o de preparar el dominio de simulación del modelo del mundo real. En general, incluye la preparación del dominio de fluidos computacional, la generación de mallas (células o elementos). Este proceso debe ser muy cuidadoso para evitar cualquier problema de simulación (podría dar un error) en la siguiente fase de la etapa de resolución. También define las condiciones de contorno y las propiedades del fluido. Este es el principal trabajo que consume tiempo de mano de obra en el flujo de trabajo de CFD, casi 50 % del tiempo se consume en las industrias. Hay muchas herramientas comerciales y de preprocesamiento de código abierto disponibles, y hemos elegido una herramienta eficiente para hacer frente a nuestro trabajo de preprocesamiento, que se describe en el Capítulo 5.

1.2. Solver

Esta área se centra en resolver las ecuaciones discretizadas (ecuaciones algebraicas) en base a modelos matemáticos. En general, las ecuaciones diferenciales parciales se discretizan mediante el uso de diferentes modelos matemáticos, como la diferencia finita, el volumen finito y los métodos de elementos finitos. Esos métodos incorporan las ecuaciones diferenciales parciales en un sistema de ecuaciones que se resolverá utilizando la computadora con el algoritmo iterativo en general. Nuevamente, hay muchas herramientas comerciales y de código abierto disponibles como solucionador (solver). Todos y cada uno de los solucionadores tienen sus propias ventajas y desventajas. En esta tesis, hemos descrito la simulación directa de elementos finitos (DFS), que es el método más avanzado (elementos eficientes de potencia computacional, consumo de tiempo y precisión) hasta ahora para resolver los problemas de turbulencia que se describe en detalle en esta tesis .

1.3. Post-procesamiento

Esta fase se asegura de que los resultados de la simulación son razonables, en comparación con los resultados experimentales, además de convocar a las personas de terceros que tienen menos conocimiento de CFD. En general, incluye, por ejemplo, representación de volumen, gráficos vectoriales y extracción de resultados de simulación (en términos de valores numéricos). También hay herramientas comerciales y de código abierto disponibles para realizar esta tarea. En esta tesis, hemos utilizado la herramienta de código abierto llamada Visita. A veces depende del volumen del archivo, puede tomar mucho tiempo procesar los datos, mientras que hemos recomendado el método eficiente para hacer este proceso usando VisIt.

2. Contenido central de la tesis

Ahora describiremos los contenidos centrales de esta tesis y sus resultados.

- Validación matemática DFS contra el estándar y complejo problema de referencia
- Comparación del DFS con otros métodos considerando el problema estándar de referencia.
- Aplicación del modelo matemático validado en el rpoblema del mundo real
- Validación de los cálculos científicos en aceleradores como GPU
- Y finalmente, recomendaciones de herramientas optimizadas para preprocesamiento y postprocesamiento, y testeo del DFS en la plataforma Cloud.

2.1. FEniCS-HPC

En esta tesis, hemos utilizado el software FEniCS-HPC. Esta es una herramienta CFD de código abierto con escalabilidad masiva paralela. Se basa en el método de elementos finitos. La principal ventaja de esta plataforma es que es muy fácil formular los modelos matemáticos cercanos al formato escrito, lo que simplifica la configuración del solucionador muy fácilmente. Y otra ventaja muy distintiva de utilizar la metodología DFS, que utiliza la condición de límite de deslizamiento para el modelo de flujo de turbulencia y la metodología de malla adaptativa, que minimiza la potencia de cálculo en comparación con otras herramientas CFD (tanto comerciales como de código abierto). Además, tiene diferentes componentes para facilitar aún más el cálculo en solucionadores iterativos, particiones de malla y herramientas para ayudar en el procesamiento previo y posterior.

2.2. Conclusiones de esta tesis por sección

La predicción del flujo separado, incluida la pérdida de un avión completo mediante la dinámica de fluidos computacional (CFD) se considera uno de los grandes desafíos que se resolverán en 2030, según NASA. Las ecuaciones no lineales de Navier-Stokes proporcionan la formulación matemática para flujo de fluidos en espacios tridimensionales. Sin embargo, todavia faltan soluciones clásicas, existencia y singularidad. Ya que el cálculo de la fuerza bruta es intratable para realizar simulación predictiva para un avión completo, uno puede usar la simulación numérica directa (DNS); sin embargo, es prohibitivamente caro ya que necesita resolver la turbulencia a escala de magnitud $Re^{\frac{9}{4}}$. Considerando otros métodos como el estadístico promedio Reynolds's Average Navier Stokes (RANS), spatial average Large Eddy Simulation (LES), y Hybrid Detached Eddy Simulation (DES), que requieren menos cantidad de grados de libertad. Todos destos métodos deben ajustarse a los problemas de referencia y, además, cerca las paredes, la malla tiene que ser muy fina para resolver las capas límite (lo cual significa que el costo computacional es muy costoso). Por encima de todo, los resultados son sensibles a, por ejemplo, parámetros explícitos en el método, la malla, etc.

Como una solución al desafío, aquí presentamos la adaptación Metodología de solución directa de FEM (DFS) con resolución numérica disparo, como una familia predictiva, libre de parámetros de métodos para flujo turbulento. Resolvimos el modelo de avión JAXA Standard Model (JSM) en número realista de Reynolds, presentado como parte del High Lift Taller de predicción 3. Predijimos un aumento de C_l dentro de un error de 5 % vs experimento, arrastre C_d dentro de 10 % error y detenga 1° dentro del ángulo de ataque.

El taller identificó un probable experimento error de pedido 10 % para los resultados de arrastre. La simulación es 10 veces más rápido y más barato en comparación con CFD tradicional o existente enfoques. La eficiencia proviene principalmente del límite de deslizamiento condición que permite mallas gruesas cerca de las paredes, orientada a objetivos control de error adaptativo que refina la malla solo donde es necesario y grandes pasos de tiempo utilizando un método de iteración de punto fijo tipo Schur, sin comprometer la precisión de los resultados de la simulación.

Como seguimiento, fuimos invitados al Quinto Taller CFD de Alto Orden, donde el enfoque fue validado para un problema de esfera en tándem (bajo Número de Reynolds flujo turbulento) en el que se coloca una segunda esfera cierta distancia aguas abajo de una primera esfera. Los resultados capturan El fenómeno del slipstream esperado, con appx. Error de 2 %. UNA comparación con los marcos de orden superior Nek500 y PyFR fue hecho. El marco PyFR ha demostrado una alta efectividad para las GPU con una malla no estructurada, lo cual es un problema difícil en este campo. Esta se logra mediante un enfoque explícito de paso de tiempo. Nuestro estudio mostró que nuestro enfoque de paso de tiempo grande permitió appx. 3 órdenes de magnitud pasos de tiempo mayores que los pasos de tiempo explícitos en PyFR, que hicieron que nuestro método más efectivo para resolver todo el problema.

También presentamos una generalización de DFS a densidad variable y validado contra el problema de referencia MARIN bien establecido. los Los resultados muestran un buen acuerdo con los resultados experimentales en forma de sensores de presión. Más tarde, usamos esta metodología para resolver dos aplicaciones en problemas de flujo multifásico. Uno tiene que ver con un flash tanque de almacenamiento de agua de lluvia (consorcio de agua de Bilbao), y el segundo es sobre el diseño de una boquilla para impresión 3D. En el agua de lluvia tanque de almacenamiento, predijimos que la altura del agua en el tanque tiene un influencia significativa sobre cómo se comporta el flujo aguas abajo de la puerta del tanque (válvula). Para la impresión 3D, desarrollamos un diseño eficiente con El flujo de chorro enfocado para evitar la oxidación y el calentamiento en la punta del boquilla durante un proceso de fusión.

Finalmente, presentamos aquí el paralelismo en múltiples GPU y el incrustado sistema de arquitectura Kalray. Casi todas las supercomputadoras de hoy tienen arquitecturas heterogéneas, como CPU GPU u otros aceleradores, y, por lo tanto, es esencial desarrollar marcos computacionales para aprovecha de ellos. Como lo hemos visto antes, se comienza a desarrollar ese CFD más tarde en la década de 1060 cuando podemos tener poder computacional, por lo tanto, Es esencial utilizar y probar estos aceleradores para los cálculos de CFD. Las GPU tienen una arquitectura diferente en comparación con las CPU tradicionales. Técnicamente, la GPU tiene muchos núcleos en comparación con las CPU que hacen de la GPU una buena opción para el cómputo paralelo.

Para múltiples GPU, desarrollamos un cálculo de plantilla, aplicado a simulación de pliegues geológicos. Exploramos la computación de halo y utilizamos Secuencias CUDA para optimizar el tiempo de computación y comunicación. La ganancia de rendimiento resultante fue de 23 % para cuatro GPU con arquitectura Fermi, y la mejora correspondiente obtenida en cuatro Las GPU Kepler fueron de 47 %.

La arquitectura Kalray está diseñada para tener poco consumo de energía. Aquí probamos el método de Jacobi con diferentes estrategias de comunicación. La idea principal es probar que la arquitectura Kalay es adecuada para los cálculos científicos. El cálculo de CFD también debe abordar la eficiencia de los cálculos y el consumo de energía.

Además, la visualización es un área crucial cuando hacemos simulaciones cientificas. Desarrollamos un marco de visualización automatizado, donde pudimos ver que la paralelización de tareas es más de 10 veces más rápido que la paralelización de datos. También hemos usado nuestro DFS en el configuración de computación en la nube para validar

la simulación contra el clúster local de simulación. Dado que el DFS consume menos potencia computacional, podemos concluir que está más optimizado para la plataforma Cloud.

Como hemos mencionado anteriormente, el pre-procesamiento puede tomar gran parte del tiempo durante el proceso de CFD. Para abordar este problema, hemos recomendado la sencilla herramienta de preprocesamiento para Soporta simulación DFS. La plataforma recomendada es un proceso eficiente y automatizado.

Contents

List of Figures

Chapter 1

Introduction

This thesis describes the mathematical prediction of turbulent incompressible Navier-Stokes equations, a generalization formulation to multiphase flow, multiphase flow with dynamics of the floating platform, benchmarking against the higher-order methods and its computational implementation and real world applications in FEniCS-HPC [8]. And also numerical simulation implementation on the parallel architecture aiming at exascale computing. FEniCS-HPC is an open-source framework for automated solution of partial differential equations (PDE) on massively parallel architectures, providing an automated evaluation of variational forms with a high-level description in mathematical notation, duality-based adaptive error control, implicit parameter-free turbulence modeling by the use of stabilized finite element methods (FEM) and strong linear scaling up to thousands of cores. FEniCS-HPC is a branch of the FEniCS framework focusing on high performance on massively parallel architectures.

The numerical analysis focuses on approximating solutions to mathematical equations that arise in science and engineering. In general, these mathematical equations are in the form of Partial Differential Equations (PDE), describing physical phenomena such as conservation, growth, etc. and quantities, such as pressure, velocity, density, and force. Solving these PDE symbolically is seldom feasible, yet it is a predictive numerical approximation.

Numerical approximation brings down the partial differential equations into a system of algebraic equations with the finite number of unknowns by using a discretization method that can be solved by numerical algebra methods. This can be automatable and effectively done on computers. There are three well known numerical methods that solve PDE: Finite Volume Method (FVM), Finite Element Method (FEM) and Finite Difference Method (FDM).

FEniCS-HPC is based on FEM, which is based on a variational form of the PDE. In the Navier-Stokes equations (NSE), if the method satisfies certain conditions on stability and consistency, the FEM solutions converge towards a weak solution to the NSE as the finite element mesh is refined [9]. Such methods are the General

Galerkin (G2) method or Direct Finite Element simulation (DFS), which is what we use and develop in the thesis.

This thesis will explain and describe the basic structure and outline of FEniCS-HPC, and the new mathematical formulations for incompressible flow and multiphase flow along with efficient parallel algorithms. We also describe how the mathematical formulation is useful in real-world applications and also give computational proof of how our approach is efficient in terms of solution accuracy and computational cost.

1.1 Motivation and Background

We will explain how one can predict the aerodynamic forces lift & drag and stall on a full aircraft at realistic Reynolds number, posed as the grand challenge problem in NASA Vision 2030, and multiphase flow simulations (for example, water dam breaking and 3D printing applications). We will show validation cases for the NASA HiLiftPW-3 full aircraft benchmark challenge, and the tandem sphere and MARIN benchmark problem.

Within the CFD community, pre-processing and post-processing consume much time. We show here, efficient post-processing technique and pre-processing technique, which can be done very quickly using the parallel processors.

As of now, we are entering into Cloud computing technology, so we will show how to create a cluster (computing cluster from the local computer) and run the simulation (FEniCS-HPC) on the Google Cloud. The simulation performance is compared against the local cluster simulation.

Finally, we show the multiple GPU stencil computations and parallel algorithms on the Kalray embedded architecture. It is quite common nowadays all the supercomputer have an accelerator such as GPUs and energy-efficient embedded hardware such as FPGA. So this work will open up the future energy-efficient and exascale computing in science and engineering for the numerical applications.

1.2 Objectives of the thesis

- The main hypothesis is that DFS for incompressible Navier-Stokes equations, predicts general aerodynamic forces, in a particular stall prediction. The main work has been focused development of the abstract methodology and FEniCS formulations in an HPC setting, with benchmarking at the highest level.

- A secondary hypothesis is that a formulation of variable-density extension in the DFS settings allows prediction in e.g. marine dam breaking, shallow water, and coastal engineering modeling.

- One objective is to compare DFS with higher-order frameworks, for low Reynolds number turbulent flow, especially to investigate the state-of-the-art of HPC and GPU performance in unstructured meshes.

- Carry out numerical simulation using multiple GPU settings, and also on the Kalray embedded architecture (energy-efficient accelerator). Thus opens up the numerical simulation to be run on the exa-scale supercomputers.

- The applications related to the multiphase flow of rainwater flash tank opening and 3D printing nozzle design.

- Recommend the parallel visualization using open-source visualization tools. Especially the remote visualization that would avoid all unnecessary ad-hoc GUI interface with remote machines. And recommend the suitable and quicker pre-processing tool for the meshing.

- Run the FEniCS-HPC simulation on the cloud platform, for example, Google-Cloud. The results will be compared against the local cluster simulation.

1.3 Work organization

This thesis work is organized as follows:

- Chapter 1 explains the introduction of the work, main thesis goals & objectives, and introduction & outline of the FEniCS-HPC framework.

- Chapter 2 explains the mathematical formulation and its validation against the experimental results. In particular, introducing the numerical tripping for the time-resolved aerodynamics simulation, multiphase flow methodology, benchmark against the higher-order methods for the low Reynolds number, and validating the dynamics of renewable energy floating platform within the multiphase flow settings.

- Chapter 3describes the real-world application based on validated mathematical modeling. It consists of aerodynamics lift and drag prediction of realistic aircraft, rainwater flash tank door opening for the shallow water modeling and 3D printing nozzle design.

- Chapter 4 recommends the tools and methodologies for pre-processing and post-processing. In general, both pre-processing and post takes much time; this Chapter 4 suggests an efficient and fast way to do that, and also explains running the FEniCS-HPC simulation on the Cloud infrastructure.

- Chapter 5 gives an overview of the high-performance computing (CPU & GPU architecture overview) and parallel programming methodologies. And also, numerical simulation on the accelerator such as GPU and in the energy-efficient computing hardware such as Kalray architecture aiming at exascale computing.

1.4 History and importance of CFD

Fluid Mechanics (FM) is an everyday part of people's lives; it can be blood flow in the human body, passenger aircraft, etc. At the beginning of the 18^{th} century, FM started to get its formal mathematical definitions by scientists like Leonhard Euler, Daniel Bernoulli, Claude-Louis Navier and Sir George Gabriel Stokes. But it progresses was not very fast in the beginning and did not take off until the advent of massive calculations with computers. In 1934 though, Leray proved the existence of weak solutions, not many classical solutions such as uniqueness exit for nonlinear Navier-Stokes equations in 3-dimensional spaces.

In the past few decades, the field of fluid mechanics has expanded dramatically. With the pervasive penetration of software, a new subfield has emerged: Computational Fluid Dynamics (CFD). In recent years, CFD has considerably replaced the experimental results; This has a significant influence on the reduced cost, reduced pollution, and efficient and innovative designs in cars, airplanes, renewable and biomedical, etc.

1.5 Components of FEniCS-HPC

FEniCS-HPC [8] is an open-source framework for the automated solution of PDEs on massively parallel architectures, providing an automated evaluation of variational forms whose description is given in a high-level mathematical notation, duality-based adaptive error control, implicit turbulence modeling using stabilized FEM and strong linear scaling up to thousands of cores [10, 11]. FEniCS-HPC is a branch of the FEniCS [12, 13] framework focusing on high performance in massively parallel architectures.

Unicorn is solver technology (models, methods, algorithms and software) with the goal of automated high-performance simulation of realistic continuum mechanics applications, such as drag or lift computation for fixed or flexible objects (FSI) in turbulent incompressible or compressible flow. The basis for Unicorn is Unified Continuum (UC) modeling [14] formulated in Euler (laboratory) coordinates, together with the General Galerkin (G2) adaptive stabilized finite element discretization described above.

FEniCS-HPC is based on a component's structure, which has many components. DOLFIN [15], which supports both C++ and python programming languages, provides the core problem solving environment to FEniCS, such as data structures, algorithms for computational meshes, and finite element assembly.

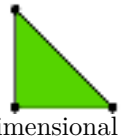
FEniCS-HPC is a problem-solving environment (PSE) for automated solution of PDE by the FEM with a high-level interface for the basic concepts of FEM: weak forms, meshes, refinement, sparse linear algebra, and with HPC concepts such as partitioning, load balancing abstracted away.

The framework is based on components with clearly defined responsibilities. A compact description of the main components follows, with their dependencies as

shown in the dependency diagram in Figure 1.1:

FIAT: Automated generation of finite element spaces V and basis functions $\phi \in V$ on the reference cell and numerical integration with FInite element Automated Tabulator (FIAT) [12, 16]

$$e = (K, V, \mathcal{L})$$



where K is a cell in a mesh $\mathcal{T}$, V is a finite-dimensional function space, $\mathcal{L}$ is a set of degrees of freedom.

FFC+UFL: Automated evaluation of weak forms in mathematical notation on one cell based on code generation with Unified Form Language (UFL) and FEniCS Form Compiler (FFC) [12, 17], using the basis functions $\phi \in V$ from FIAT. For example, in the case of the Laplacian operator,

$$A_{ij}^K = a_K(\phi_i, \phi_j) = \int_K \nabla\phi_i \cdot \nabla\phi_j dx = \int_K lhs(r(\phi_i, \phi_j)dx)$$

where A^K is the element stiffness matrix and $r(\cdot, \cdot)$ is the weak residual.

DOLFIN-HPC: Automated high performance assembly of weak forms and interface to linear algebra of discrete systems and mesh refinement on a distributed mesh $\mathcal{T}_\Omega$ [18].

$A = 0$
`for all cells` $K \in \mathcal{T}_\Omega$
A `+=` A^K

$Ax = b$

Unicorn: Automated Unified Continuum modeling with Unicorn choosing a specific weak residual form for incompressible balance equations of mass and momentum with example visualizations of aircraft simulation below left and turbulent FSI in vocal folds below right [19].

$$r_{UC}((v,q),(u,p)) = (v, \rho(\partial_t u + (u\cdot\nabla)u) + \nabla\cdot\sigma - g) + (q, \nabla\cdot u) + LS((v,q),(u,p))$$

where LS is a least-squares stabilizing term described in [20].

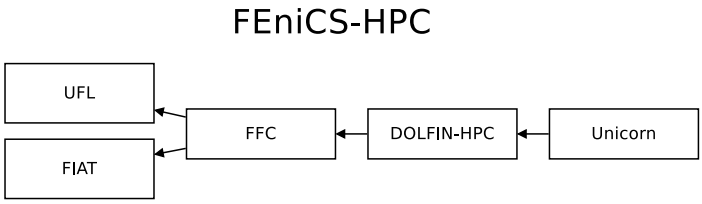


Figure 1.1: FEniCS-HPC component dependency diagram.

A user of FEniCS-HPC writes the weak forms in the UFL language, compiles it with FFC, and includes it in a high-level "solver" written in C++ in DOLFIN-HPC to read in a mesh, assemble the forms, solve linear systems, refine the mesh, etc. The Unicorn solver for adaptive computation of turbulent flow and FSI is developed as part of FEniCS-HPC.

FEniCS-HPC solves the PDE in distributed memory architectures (at the moment only with MPI and PGAS), where users can define PDE at a higher level mathematical notation of FEM in weak/variational form. This is compiled into low-level assembly functions with the help of FEniCS components such as FEniCS Form Compiler (FFC), The Unified Form Language (UFL) and FInite element Automatic Tabulator (FIAT).

FEniCS-HPC focuses on two components: DOLFIN-HPC and Unicorn. DOLFIN-HPC has specific development supporting good parallel scaling in the linear algebra interface to PETSc, parallel mesh distribution, refinement, and load balancing on the parallel computer architecture environment [21]. Unicorn provides a solving environment for Unified Continuum modeling: Direct FEM Simulation for a general continuum, e.g. incompressible NS equation or fluid-structure interaction with goal-oriented error control estimation, robust stabilization, slip boundary conditions, high-level mathematical abstraction for time-dependent PDE such as fluid-structure interaction (FSI) problems. Figure 1.1 shows the typical workflow of the FEniCS-HPC.

Chapter 2

Mathematical formulation and validation

In this chapter, we will develop the mathematical formulations and validations.

2.1 Direct FEM Simulation

The Direct FEM Simulation (DFS) methodology is based on directly solving the continuum model, e.g. Navier-Stokes equations, without an explicit turbulence model, with a robust finite element method consisting of: residual-based FEM with stabilization, a posteriori error estimation, and a goal-oriented adaptive error control algorithm.

2.1.1 The cG(1)cG(1) method

As the basic model for incompressible Newtonian fluid flow, we consider the NSE with constant kinematic viscosity $\nu > 0$, enclosed in $\Omega \subset \mathbb{R}^3$, with boundary Γ, over a time interval $I = [0, T]$:

$$\begin{cases} \dot{\vec{u}} + (\vec{u} \cdot \nabla)\vec{u} + \nabla p - 2\nu \nabla \cdot \epsilon(\vec{u}) = f, & (\vec{x}, t) \in \Omega \times I, \\ \nabla \cdot \vec{u} = 0, & (\vec{x}, t) \in \Omega \times I, \\ \vec{u}(\vec{x}, 0) = \vec{u}^0(\vec{x}), & \vec{x} \in \Omega, \end{cases} \tag{2.1}$$

with $\vec{u}(\vec{x}, t)$ the velocity vector, $p(\vec{x}, t)$ the pressure, $\vec{u}^0(\vec{x})$ the initial data and $f(\vec{x}, t)$ a body force. Moreover, $\sigma_{ij} = 2\nu\epsilon_{ij}(\vec{u}) - p\delta_{ij}$ is the stress tensor, with the strain rate tensor $\epsilon_{ij}(\vec{u}) = 1/2(\partial u_i/\partial x_j + \partial u_j/\partial x_i)$, and δ_{ij} the Kronecker delta. The relative importance of viscous and inertial effects in the flow is determined by the Reynolds number $Re = \vec{U}L/\nu$, where $\vec{U}$ and L are characteristic velocity and length scales respectively.

The cG(1)cG(1) method is based on the continuous Galerkin method cG(1) in space and time. With cG(1) in time, the trial functions are continuous, piecewise

linear and the test functions piecewise constant. cG(1) in space corresponds to both test functions and trial functions are being continuous and piecewise linear.

Let $0 = t_0 < t_1 < ... < t_N = T$ be a sequence of discrete time steps, with associated time intervals $I_n = (t_{n-1}, t_n)$ of length $k_n = t_n - t_{n-1}$, and let $W \subset H^1(\Omega)$ be a finite element space consisting of continuous, piecewise linear functions on a tetrahedral mesh $\mathcal{T} = \{K\}$ of mesh size $h(\vec{x})$, with $W_{\vec{w}}$ the functions $\vec{v} \in W$ satisfying the Dirichlet boundary condition $\vec{v}|_\Gamma = \vec{w}$.

We seek $\hat{\vec{U}} = (\vec{U}, P)$, continuous piecewise linear in space and time, and the cG(1)cG(1) method for the NSE with homogeneous Dirichlet boundary conditions reads: for $n = 1, ..., N$ find $(\vec{U}^n, P^n) \equiv (\vec{U}(t_n), P(t_n))$, with $\vec{U}^n \in V_0 \equiv [W_0]^3$ and $P^n \in W$, such that:

$$\begin{aligned}&((\vec{U}^n - \vec{U}^{n-1})k_n^{-1} + \bar{\vec{U}}^n \cdot \nabla \bar{\vec{U}}^n, \vec{v}) + (2\nu\epsilon(\bar{\vec{U}}^n), \epsilon(\vec{v})) - (P^n, \nabla \cdot \vec{v}) \\ &+ (\nabla \cdot \bar{\vec{U}}^n, q) + SD_\delta^n(\bar{\vec{U}}^n, P^n; \vec{v}, q) = (f, \vec{v}), \quad \forall \hat{\vec{v}} = (\vec{v}, q) \in V_0 \times W,\end{aligned} \tag{2.2}$$

where $\bar{\vec{U}}^n = \frac{1}{2}(\vec{U}^n + \vec{U}^{n-1})$ is piecewise constant in time over I_n, with the stabilizing term

$$\begin{aligned}&SD_\delta^n(\bar{\vec{U}}^n, P^n; \vec{v}, q) \equiv \\ &\quad (\delta_1(\bar{\vec{U}}^n \cdot \nabla \bar{\vec{U}}^n + \nabla P^n - f), \bar{\vec{U}}^n \cdot \nabla \vec{v} + \nabla q) + (\delta_1 \nabla \cdot \bar{\vec{U}}^n, \nabla \cdot \vec{v}),\end{aligned} \tag{2.3}$$

and

$$\begin{aligned}(\vec{v}, \vec{w}) &= \sum_{K \in \mathcal{T}} \int_K \vec{v} \cdot \vec{w}\, dx, \\ (\epsilon(\vec{v}), \epsilon(\vec{w})) &= \sum_{i,j=1}^{3} (\epsilon_{ij}(\vec{v}), \epsilon_{ij}(\vec{w})),\end{aligned}$$

with the stabilization parameter $\delta_1 = \kappa_1 h$, where κ_1 is a positive constant of unit size. We chose a time step size $k_n = C_{CFL} \min_{\vec{x} \in \Omega} h/|\vec{U}^{n-1}|$, with C_{CFL} typically in the range $[0.5, 20]$. The resulting non-linear algebraic equation system is solved with a robust Schur-type fixed-point iteration method [22].

2.1.2 The Adaptive Algorithm

A simple description of the adaptive algorithm, starting from $i = 0$, reads:

1. For the mesh $\mathcal{T}_i$: solve the primal and (linearized) dual problems for the primal solution $(\vec{U}, P)$ and the dual solution (Φ, Θ).
2. Compute the quantity $\mathcal{E}_K$ for any cell K of $\mathcal{T}_i$. If $\sum_{K \in \mathcal{T}_i} \mathcal{E}_K < TOL$ and then stop, else:

3. Mark 5% of the elements with highest $\mathcal{E}_K$ for refinement.

4. Generate the refined mesh $\mathcal{T}_{i+1}$, and go to 1.

Here, $\mathcal{E}_K$ is the *error indicator* for each cell K, which we describe in Section 2.1.3. For now, it suffices to say that $\mathcal{E}_K$ is a function of the residual of the NSE and the solution to a linearized dual problem. The formulation of the dual problem includes the definition of a *target functional* for the refinement, which usually enters the dual equations as a boundary condition or as a volume source term. This functional should be chosen according to the problem we are solving. In other words, one needs to ask the right question in order to obtain the correct answer from the algorithm. In this chapter, our target functional is chosen to be the mean value in time of the aerodynamic forces.

The dual problem can be written as (see [23] for more details):

$$\begin{cases} -\dot{\vec{\varphi}} - (\vec{u}\cdot\nabla)\vec{\varphi} + \nabla\vec{U}^{\top}\vec{\varphi} + \nabla\theta - \nu\Delta\vec{\varphi} = \psi_1 & (\vec{x},t) \in \Omega \times I \\ \nabla\cdot\vec{\varphi} = \psi_2 & (\vec{x},t) \in \Omega \times I \\ \vec{\varphi} = \psi_3 & (\vec{x},t) \in \Gamma \times I \\ \vec{\varphi}(\cdot,T) = \psi_4 & \vec{x} \in \Omega, \end{cases} \tag{2.4}$$

where we find that the structure is similar to the primal NSE equations, except that the adjoint problem is linear, the transport is backward in time, and that we have a reaction term $(\nabla\vec{U}^{\top}\vec{\varphi})_j = U_{,j}\cdot\vec{\varphi}$, that is not present in the primal NSE.

The only other input required from the user is an initial discretization of the geometry, $\mathcal{T}_0$. Since our method is designed for tetrahedral meshes that do not require any special treatment of the near-wall region (no need for a boundary-layer mesh), the initial mesh can be easily created with any standard mesh generation tool.

2.1.3 A posteriori error estimate for cG(1)cG(1)

The a posteriori error estimate is based on the following theorem (for a detailed proof, see chapter 30 in [9]):

Theorem 1 *If $\hat{\vec{U}} = (\vec{U}, P)$ solves (2.2), $\hat{\vec{u}} = (\vec{u}, p)$ is a weak NSE solution, and $\hat{\vec{\varphi}} = (\vec{\varphi}, \theta)$ solves an associated dual problem with data $M(\cdot)$, then we have the following a posteriori error estimate for the target functional $M(\hat{\vec{U}})$ with respect to the reference functional $M(\hat{\vec{u}})$:*

$$|M(\hat{\vec{u}}) - M(\hat{\vec{U}})| \le \sum_{n=1}^{N} \Bigg[\int_{I_n} \sum_{K\in\mathcal{T}_i} |R_1(\vec{U},P)_K| \cdot \omega_1 \; dt$$
$$+ \int_{I_n} \sum_{K\in\mathcal{T}_i} |R_2(\vec{U})_K| \; \omega_2 \; dt + \int_{I_n} \sum_{K\in\mathcal{T}_i} |SD_\delta^n(\hat{\vec{U}};\hat{\vec{\varphi}})_K| \; dt \Bigg] =: \sum_{K\in\mathcal{T}_i} \mathcal{E}_K$$

with

$$R_1(\vec{U}, P) = \dot{\vec{U}} + (\vec{U} \cdot \nabla)\vec{U} + \nabla P - 2\nu\nabla \cdot \epsilon(\vec{u}) - f,$$
$$R_2(\vec{U}) = \nabla \cdot \vec{U}, \tag{2.5}$$

where $SD^n_\delta(\cdot;\cdot)_K$ *is a local version of the stabilization form (2.3), and the stability weights are given as*

$$\omega_1 = C_1 h_K |\nabla\vec{\varphi}|_K,$$
$$\omega_2 = C_2 h_K |\nabla\theta|_K,$$

where h_K *is the diameter of element* K *in the mesh* $\mathcal{T}_i$*, and* $C_{1,2}$ *represent interpolation constants. Moreover,* $|w|_K \equiv (\|w_1\|_K, \|w_2\|_K, \|w_3\|_K)$*, with* $\|w\|_K = (w, w)^{1/2}_K$*, and the dot denotes the scalar product in* $\mathbb{R}^3$*.*

For simplicity, here, we assumed that the time derivatives of the dual variables $\hat{\varphi} = (\vec{\varphi}, \theta)$ can be bounded by their spatial derivatives. Using Theorem 1, we can understand the adaptive algorithm. As mentioned above, the error indicator, $\mathcal{E}_K$, is a function of the residual of the NSE and the solution of a linearized dual problem (a detailed formulation of the dual problem is given in Chapter 14 in [9]). Thus, on a given mesh, we must first solve the NSE to compute the residuals, $R_1(\vec{U}, P)$ and $R_2(\vec{U})$, and then a linearized dual problem to compute the weights multiplying the residuals, ω_1 and ω_2. With that information, we are able to compute $\sum_{K\in\mathcal{T}_i} \mathcal{E}_K$ and check it against the given stop criterion. This procedure of solving the forward and backward problems for the NSE is closely related to an optimization loop and can be understood as the problem of finding the "optimal mesh" for a given geometry and boundary conditions, id est, the mesh with the least possible number of degrees of freedom for computing $M(\hat{\vec{u}})$ within a given degree of accuracy.

2.1.4 The Do-nothing Error Estimate and Indicator

To minimize the loss of sharpness, we also investigated an approach where the weak form is used directly in a posteriori error estimates without integration by parts to the strong form using the Cauchy-Schwarz inequality and interpolation estimates. Here, we refer to this direct form of a posteriori error representation by duality as the "do-nothing" approach.

In terms of the exact adjoint solution $\hat{\vec{\varphi}}$, the output error with respect to a weak solution $\hat{\vec{u}}$ can be represented as

$$|M(\hat{\vec{u}}) - M(\hat{\vec{U}})| = |(R(\hat{\vec{U}}), \hat{\vec{\varphi}})| = |\sum_{K\in\mathcal{T}_i} (R(\hat{\vec{U}}), \hat{\vec{\varphi}})_K| \tag{2.6}$$

This error representation involves no approximation or inequalities. We thus refer to the following error indicator based on the representation as the *do-nothing error indicator*:

$$e^K \equiv (R(\hat{\vec{U}}), \hat{\vec{\varphi}})_K \tag{2.7}$$

A computable estimate and an error indicator are again based on the computed approximation $\hat{\vec{\varphi}}_h$ of the dual solution:

$$|M(\hat{\vec{u}}) - M(\hat{\vec{U}})| \approx |(R(\hat{\vec{U}}), \hat{\vec{\varphi}}_h)| \tag{2.8}$$

$$e_h^K \equiv (R(\hat{\vec{U}}), \hat{\vec{\varphi}}_h)_K \tag{2.9}$$

where we may lose reliability of the global error estimate by the Galerkin orthogonality property, which states that the $(R(\hat{\vec{U}}), \hat{\vec{\varphi}}_h)$ vanishes for a standard Galerkin finite element method if $\hat{\vec{\varphi}}_h$ is chosen in the same space as the test functions. Although, in the setting of a stabilised finite element method, this may not be the case, see [24].

2.1.5 Boundary layers: medium Reynolds number flow

If the Reynolds number is less than $Re = 10^5$, then we choose a no-slip boundary condition, since it's tractable to resolve it with the mesh. This has been verified for numerous cases, such as cube [25], rectangular cylinder [26], sphere [27] and circular cylinder [28]. In all these cases, the solution converges towards the reference output quantities such as drag, lift and pressure output with less degree of freedom when compared to standard LES methods, which are based on manual meshing.

2.1.6 Boundary layers: higher Reynolds number flow

In our work on high Reynolds number turbulent flows [29–31] we chose a skin friction stress as the wall layer model. That is, we appended the NSE with the following boundary conditions:

$$\vec{u} \cdot \vec{n} = 0, \tag{2.10}$$

$$\beta \vec{u} \cdot \tau_k + \vec{n}^T \sigma \tau_k = 0, \quad k = 1, 2, \tag{2.11}$$

for $(\vec{x}, t) \in \Gamma_{solid} \times I$, with $\vec{n} = \vec{n}(\vec{x})$ an outward unit normal vector, and $\tau_k = \tau_k(\vec{x})$ orthogonal unit tangent vectors of the solid boundary Γ_{solid}. We used matrix notation with all vectors $\vec{v}$ being column vectors and the corresponding row vector as denoted by $\vec{v}^T$.

With skin friction boundary conditions, the rate of kinetic energy dissipation in cG(1)cG(1) makes a contribution to the form

$$\sum_{k=1}^{2} \int_0^T \int_{\Gamma_{solid}} |\beta^{1/2} \bar{\vec{U}} \cdot \tau_k|^2 \, ds \, dt, \tag{2.12}$$

from the kinetic energy, which is dissipated as friction in the boundary layer. For high Re, we modeled $Re \to \infty$ by $\beta \to 0$, so that the dissipative effect of the

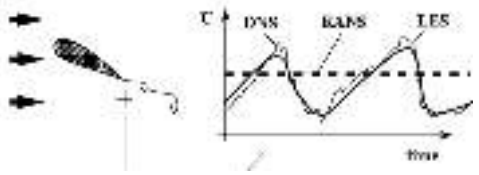


Figure 2.1: Illustration of the difference between DNS, LES and RANS [source from Nicoud Franck, dec 2007].

boundary layer vanishes with large Re. In particular, we found that a small β does not influence the solution [29]. For the present simulations, we used the approximation $\beta = 0$, which can be expected to be a good approximation for real high-lift configurations, where Re is very high.

2.2 Time resolved adaptive direct FEM simulation

To predict separated flow, including the stall of a full aircraft with Computational Fluid Dynamics (CFD) is of key importance to society, it is considered by NASA as one of the grand challenges to be solved by 2030 [1]. This is a turbulent flow problem. Brute-force computation is intractable, therefore, to do predictive simulation for a full aircraft, one can use Direct Numerical Simulation (DNS), but it is prohibitively expensive as it needs to resolve the turbulent scales of order $Re^{\frac{9}{4}}$. The scales are extremely small when compared to the total length scale of the aircraft, requires very small elements in an enormous domain. It is thus not feasible to compute, even on the largest supercomputers today.

There are other methods that require less number of degrees of freedom, that is, statistical average Reynolds's Average Navier Stokes (RANS), spatial average Large Eddy Simulation (LES) and hybrid Detached Eddy Simulation (DES). All of these methods have to be tuned to benchmark problems, and, moreover, also near the walls, the mesh has to be very fine to resolve boundary layers (which means the computational cost is very expensive). Above all, the results are sensitive to, for e.g. explicit parameters in the method, the mesh, etc. Figure 2.1 shows the basic illustration of DNS, LES, and RANS.

In contrast to the statistical averages of RANS and the filtered solutions of LES, our simulation method is based on the computational approximation of weak solutions to the Navier-Stokes equations (NSE), that satisfy the NSE in variational form integrated against a class of test functions.

Finite element methods (FEM) are based on a variational form of the NSE, and if the method satisfies certain conditions on stability and consistency, the FEM solutions converge towards a weak solution to the NSE as the finite element mesh is refined [9]. We refer to such FEM as a General Galerkin (G2) method or a Direct Finite Element simulation (DFS).

The resolution in DFS is set by the mesh size, and no turbulence model is introduced. The dissipation of turbulent kinetic energy in under-resolved parts of the flow is provided by the numerical stabilization of G2 in the form of a weighted least squares method based on the residual of NSE.

The mesh is adaptively constructed based on a posteriori estimation of the error in the chosen goal or target functionals, such as drag and lift forces. The a posteriori error estimates take the form of a residual weighted by the solution of an adjoint problem, which is computed separately using a similar stabilized FEM method [9]. The adaptive algorithm starts from a coarse mesh, which is locally refined during each iteration based on the a posteriori error estimates.

We used a free slip boundary condition as a model for high Reynolds number turbulent boundary layers with small skin friction. This means that boundary layers are left unresolved and that no boundary layer mesh is needed.

DFS is a unique approach as it does not require any explicit turbulence modeling and boundary layer model. The computational mesh is refined based on the posteriori error estimation using an adjoint technique. There is no need for a boundary layer mesh and no need to assume the flow behavior before the simulation and refine the mesh manually in the fluid domain. The boundary layer is defined by the wall stress in terms of skin friction. As the Reynolds number goes to infinity, the skin friction goes to zero. The consequence is the free slip boundary condition, which facilitates an enormous computational efficiency.

2.2.1 Problem description of JSM full aircraft

The HiLiftPW-3 workshop provides the CAD model of the JAXA Standard Model (JSM). Figure 3.6 shows the pressure sensor locations on the JSM aircraft and figure 2.3 shows the JSM model of curvature resolved mesh and volume mesh (about 2 5M cells) as an adaptive mesh for the starting simulation.

2.2.2 Numerical tripping

Typically, in the field and our previous work, computations are noise-free, which means boundaries are very smooth, and there are no vibrations. However, in reality, for example, in an airplane this is not the case, there is noise in the air, vibrations and rough surfaces.

Recently, the noise has been introduced in the DNS community, which has shown more uniform results [32]. A similar approach is used in experiments, where tripping is introduced for consistent results.

Inspired by this, we have introduced a similar approach in DFS, by introducing a volume force on the bounding box of the aircraft. To prevent the noise from dominating the solution, we choose the magnitude to 5% of pressure gradient as white noise.

In the HiLiftPW-2, we could not predict stall with DFS. With the introduction of numerical tripping [33] we were able to predict stall in HiLiftPW-3; this appears

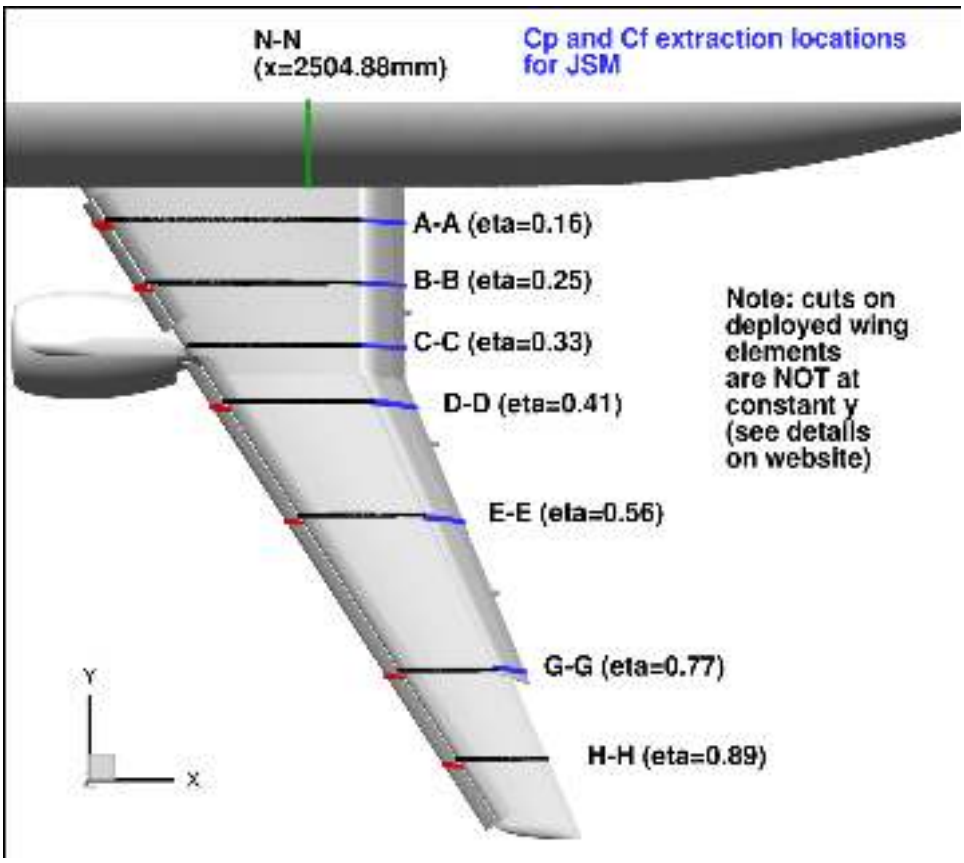


Figure 2.2: Pressure sensor layout on JSM configuration.

to have been the missing ingredient. With stall prediction, we thus have the full resolution of the NASA Vision 2030 challenge.

2.2.3 Aerodynamic Forces

The aerodynamic forces with unit velocity are computed as follows:

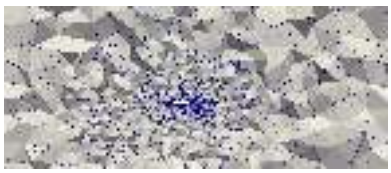
$$F = \frac{1}{|I|} \int_I \int_{\Gamma_a} p\vec{n}\, dsdt, \tag{2.13}$$

with, Γ_a as left half-boundary of the aircraft. The drag and lift coefficients are simply the x and y components of F.

(a) Surface mesh.

(b) Pylon.

(c) Volume mesh.



(d) Wing slice.

Figure 2.3: JSM aircraft model with starting mesh for the adaptive method.

2.2.4 Validation

HiLiftPW-3 provided us the aerodynamics forces for angles 4.36° , 10.58° , 18.58°, 21.57° and 22.58°. We compared all the angle results with simulation. In particular, we compared the aerodynamic forces both with and without tripping for the angles 18.58° and 22.58°. Figure 2.4 shows the comparison between experimental and simulation. We used a "do-nothing" adaptive approach that refines the mesh based on solving a primal and dual problem using the posteriori error estimation. So, in figure 2.4, the results are compared to the optimal mesh solution (after the mesh convergence).

We observed both C_l and C_d matches experimental results with 5% and 10% error. We also closely predicted stall in the angles of 18.58°, 21.57° and 22.58°. We also saw that stall is happening in 18.58°, 21.57° within the 1° error. Our tripping approach does not affect the solution at 18.58°, which is the maximum lift and maximum pre-stall angle. Whereas on 22.58°, we observed that stall occurs with large separation when compared to the non-tripped approach. This concludes that DFS with tripping facilitates stall prediction for realistic aircraft at a realistic Reynolds number.

More detail comparison (quantitative and qualitative) of lift and drag against the experimental data is explained in the application Chapter 3.

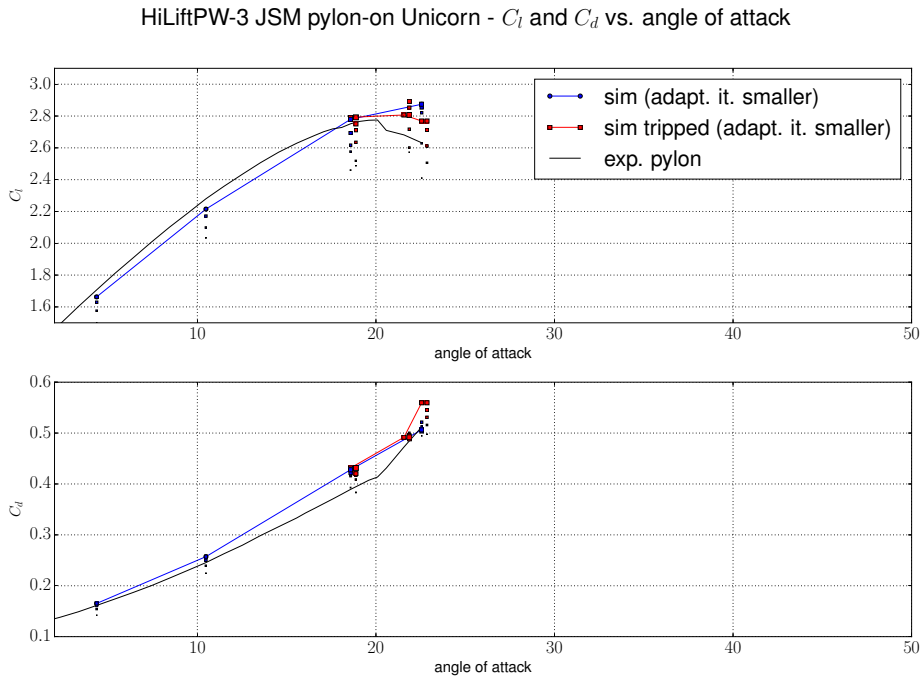


Figure 2.4: Lift coefficient, C_l, and drag coefficient, C_d, versus the angle of attack, α, for the different meshes from the iterative adaptive method.

2.3 Multi Phase Flow

In this section, we will describe multiphase flow formulation. Multiphase flow is essential modeling in science and engineering, for example, chemical industries and renewable marine engineering. We introduced a variable-density incompressible Navier-Stokes formulation in the DFS methodology, where DFS is implemented in FEniCS-HPC.

2.3.1 Mathematical model

We modeled the problem using the primitive incompressible Navier-Stokes equations with variable density ρ:

$$
\begin{aligned}
R(\hat{u}) &= \begin{cases} \rho(\partial_t u + (u \cdot \nabla)u) + \nabla p - \nu \Delta u - \rho g = 0 \\ \partial_t \rho + (u \cdot \nabla)\rho = 0 \\ \nabla \cdot u = 0 \end{cases} \\
\hat{u} &= (u, p, \rho)
\end{aligned}
$$

By using a few parameter-free stabilized finite element method (FEM), we were not introducing any explicit parameterization or modeling, aside from the slip model of the boundary layer and we thus expected the simulations to be *predictive* if the mesh is fine enough to control the computational error.

2.3.2 Direct FEM cG(1)cG(1) for variable-density

In a cG(1)cG(1) method [34], we sought an approximate space-time solution $\hat{U} = (D, U, P)$ (with D the discrete density ρ) that is continuous piecewise linear in space and time (equivalent to the implicit Crank-Nicolson method). With I a time interval with sub intervals $I_n = (t_{n-1}, t_n)$, W^n a standard spatial finite element space of continuous piecewise linear functions, and W_0^n the functions in W^n, which are zero on the boundary Γ, the cG(1)cG(1) method for variable-density incompressible flow with homogeneous Dirichlet boundary conditions for the velocity takes the following form: for $n = 1, ..., N$, find $(D^n, U^n, P^n) \equiv (D(t_n), U(t_n), P(t_n))$ with $D^n \in W^n$, $U^n \in V_0^n \equiv [W_0^n]^3$ and $P^n \in W^n$, such that

$$
\begin{aligned}
&r(\hat{U}, \hat{v}) = (D((U^n - U^{n-1})k_n^{-1} + (\bar{U}^n \cdot \nabla)\bar{U}^n), v) + (2\nu\epsilon(\bar{U}^n), \epsilon(v)) \\
&- (P, \nabla \cdot v) - (Dg, v) + (\nabla \cdot \bar{U}^n, q) + (D^n - D^{n-1})k_n^{-1} + (\bar{U}^n \cdot \nabla)\bar{D}^n), v) \\
&+ LS(D^n, \bar{U}^n, P^n) + SC(D^n, \bar{U}^n, P^n) = 0, \ \forall \hat{v} = (z, v, q) \in W^n \times V_0^n \times W^n
\end{aligned} \tag{2.14}
$$

where $\bar{U}^n = 1/2(U^n + U^{n-1})$ is piecewise constant in time over I_n and LS and SC are least-squares and shock-capturing stabilizing term described in [34].

Here, we also add an experimental phase separation term for the form $(\rho_{air} - \rho)(\rho_{water} - \rho)$.

2.3.3 Validation

Figure 2.5 shows a marine engineering dam breaking wave impact setup, which is the MARIN benchmark [35] problem. Water is stored in a rectangular box; the release of the water impacts the box. On the box, there are pressure sensors, which are compared to the simulation results.

Since for the variable density method, we do not have an adaptive approach yet; we use uniform refinement. We have approximately uniform mesh size and ca. 2 million vertices.

In Figure 2.6, we see water isosurface and contour of water density. Figure 2.7 shows the pressure comparison with simulation and experiment, and pressure

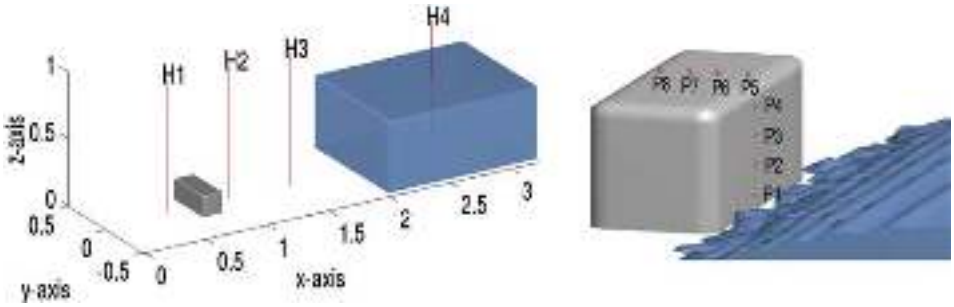


Figure 2.5: Layout of dam breaking benchmark settings [source from K.M.T. Kleefsman et al.,2005].

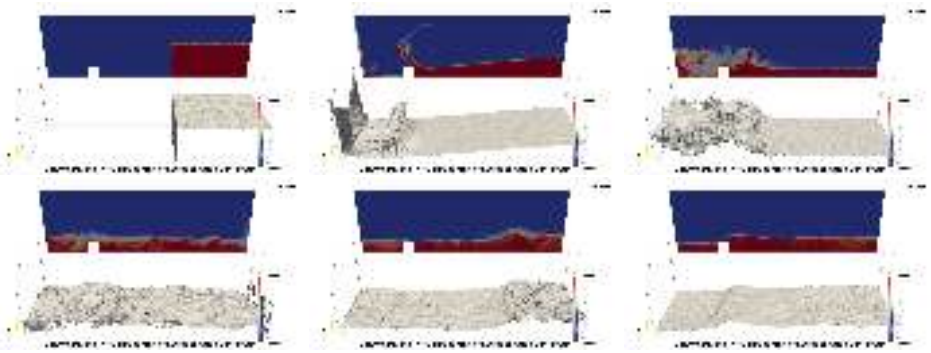


Figure 2.6: Density slice and isosurface for $\rho = 0.5$ at $t = 0, 1, 2, 3, 4, 5s$.

values are compared at location P1 and P7. We notice that there is good agreement between simulation and experimental values for the phase separation variant, and worse agreement without the phase separation term. Later this approach is adapted to two applications: one is a flash rainwater tank door opening and the second application is 3D printing. These are well explained in Chapter 3.

2.4 Tandem sphere

In this section, we will describe tandem sphere simulations in the Fifth AIAA High Order CFD Workshop and compare our DFS results with higher-order methods, in particular, PyFR and Nek5000. Higher-order methods are defined as giving a better order of accuracy than 2 for a smooth solution. Another possible advantage of higher-order formulations are more compact computation, which can improve the communication performance in an HPC setting. For further details of higher-order

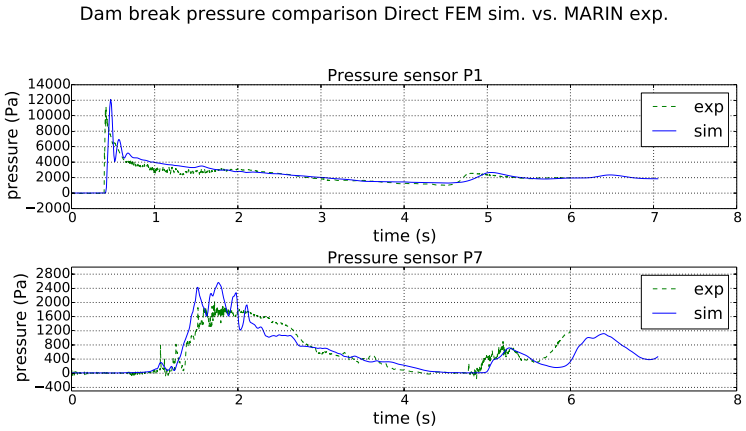


Figure 2.7: Pressure over time for simulation and experiment for the sensors P1 and P7 in the MARIN benchmark.

methods, please refer to [36–38].

The tandem sphere problem is a complex unsteady multi-scale flow under turbulent low Reynolds and Mach number. Here, we validate C_d in the DFS setting. The problem setup information is as follows:

- The diameter of the sphere is D and and another sphere is kept at $10D$ distance, with the angle of attack is 0°
- Reynolds number is $Re_D = 3900$
- Free stream temperature is $T_\infty = 300K$ and density is $\rho_\infty = 1.225 \frac{kg}{m^3}$

We constructed the starting coarse tetrahedral mesh with appx. 100k vertices, resolving the curvature of the sphere boundaries and we did not resolve any boundary layer or locally refine the mesh in the fluid domain, that is, both in upstream and downstream sides of the corresponding spheres. Figure 2.8 shows the initial mesh for the DFS simulation. In contract Figures 2.9 and 2.10, show how the higher-order CFD solvers manually refine the mesh at the boundary layer and refine the mesh along the flow path upstream and downstream of the spheres.

For low Reynolds number, i.e. less than $Re = 10^5$, we use the no-slip boundary condition. In this regime the skin friction is still significant, and boundary layers can be resolved while still keeping the computational efficiency high.

2.4.1 PyFR

PyFR is an open-source Python-based software framework for solving advection-diffusion type problems on parallel architectures using the Flux Reconstruction

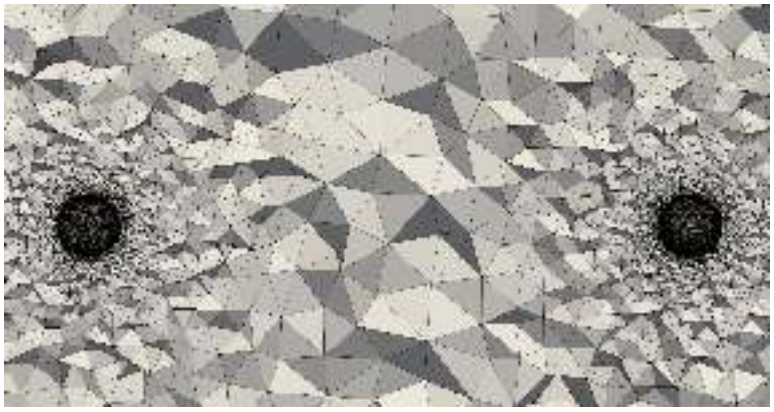


Figure 2.8: Initial mesh for the DFS simulation with appx. 100k vertices.

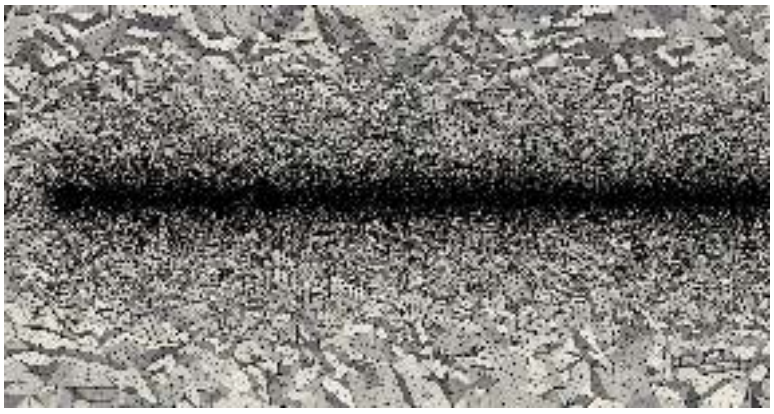


Figure 2.9: M1P3 refined mesh at down stream side of sphere 1 and 2 and refined mesh at upstream side of sphere 2.

Figure 2.10: M1P3 boundary layer is refined manually.

Mesh order	Points	Elements	hmin	hmax	CPU core hours
TandemSpheresTetMesh0.msh	16738	100516	0.19771	104.458	771.2
TandemSpheresTetMesh0P2.msh	133893	100516	0.19771/2	104.458	4958.8
TandemSpheresTetMesh0P3.msh	451792	100516	0.19771/3	104.458	9600
TandemSpheresTetMesh1.msh	41379	248707	0.12839	69.757	2320
TandemSpheresTetMesh1P2.msh	331084	248707	0.12839/2	69.757	8710.4
TandemSpheresTetMesh1P3.msh	1117069	248707	0.12839/3	69.757	14400

Table 2.1: PyFR mesh statistics and computational time.

approach of Huynh [39]. It solves the Euler and Navier Stokes equations using the mixed unstructured grids (Triangles, Quadrilaterals, Hexahedra, Prisms, Tetrahedra, Pyramids). It supports both 2D and 3D problems. Table 2.1 shows the few mesh statistics and its computational time. Listing 2.4.1 shows the setting of the PyFR for the computation. Although PyFR supports GPU, we have only tested the PyFR with CPU, because at the moment, DFS runs only on the CPU. So we thought it would be fair to compare with the CPUs.

```
%flow conditions:
%================
Mach number              = 0.1
Raynolds number          = 3900
Free stream temperature  = 300 K
Density                  = 1.225 kg/m3
Angle of attach          = 0 degree

%Boundary conditions:
```

```
%====================
%Far field      : characterictic
%Sphere surface : adiabatic wall


%Naviour stokes equations:
Specific heats = 1.4
Prandel number = 0.72

%Reference
%=========
%https://how5.cenaero.be/sites/how5.cenaero.be/files/
    CS1_TandemSpheres_0.pdf


%PyFR_BC
%=======


%1) Find the static pressure for the computation

%Pd  -> Dynamic pressrue
%u   -> Free stream velocity
%rho -> Density of the fluid

Pd = 0.5*rho*u*u
Pd = 0.5*1.225*1*1
Pd = 0.6125

Ps = Static pressure

Pd = (density * velocity^2) / 2 = (specific_heat * Ps * Mach_number^2)
     / 2
Pd = (1.225*1*1)/2
Pd = 0.6125
Ps = Pd/(0.5*specific_heat * Mach_number^2)
Ps = Pd/(0.5*1.4*0.1*0.1)

Ps = 87.5 kg/m^3

%Reference
%=========
%https://en.wikipedia.org/wiki/Dynamic_pressure
%https://www.grc.nasa.gov/www/BGH/isentrop.html


%PyFR.ini
%========

[constants]
gamma = 1.4 ;specific heat
mu = 0.000314 ;1.225/3900.0 ;viscosity
Uin = 1.0 ; inlet velocity at x direction
Vin = 0.0 ; inlet velocity at y direction
Win = 0.0 ; inlet velocity at z direction
```

```
Rho = 1.225 ; density
Pr = 0.72 ; prandle number
Pc = 87.5 ; pressure Pd =  0.6125 = 0.5*gamma*Pc*(0.1*0.1)

[solver-time-integrator]
formulation = std
scheme = rk34 ; this works!!! and no time rejection, it choses the
    best time and stays
%scheme = rk45 gives more time step rejection, it works and when it
    tries to find a advanced time it crashes
%scheme = rk4 it crashes from the begining
controller = pi
tstart = 0.0
tend = 200.0
dt = 0.148282  ; hmin 0.19771 and dt = 0.75*(hmin/P);  OPTION 3 time
    step
atol = 1e-6
rtol = 1e-6
errest-norm = l2
safety-fact = 0.9
min-fact = 0.5
max-fact = 3.0

[solver-interfaces]
riemann-solver = rusanov
ldg-beta = 0.5
ldg-tau = 0.1

[solver-interfaces]
riemann-solver = rusanov
ldg-beta = 0.5
ldg-tau = 0.1

[solver-elements-tet]
soln-pts = shunn-ham

[solver-interfaces-tri]
flux-pts = williams-shunn

[soln-plugin-writer]
dt-out = 1
basedir = .
basename = TandemSpheresTetMesh0P1{t:.2f}

[soln-plugin-fluidforce-frontsphere]
nsteps = 10
file = frontsphere-forces.csv
header = true

[soln-plugin-fluidforce-backsphere]
nsteps = 10
file = backsphere-forces.csv
header = true

[soln-plugin-nancheck]
```

```
nsteps = 10

[soln-plugin-dtstats]
flushsteps = 10
file = dtstats.csv
header = true
```

2.4.1.1 Validation:Drag C_d over time with adaptive mesh

Figure 2.12 shows the drag C_d evaluation over time with the adaptive mesh of tandem sphere. As we can see in Figure 2.12, we predicted drag C_d very close to the experimental results for a single sphere (2 % error). For the second sphere, we predicted a slightly reduced drag compared to the first sphere. We interpret this as the "slipstream" effect, as the wake from the first sphere reduces the upstream pressure on the second sphere. This is a well-known concept, and it is effectively being applied in trucks, train, and smart driving to reduce the drag [40–43]. Figure 2.11 shows the adaptive mesh at 24_{th} iteration. Here, we can see that the downstream side of the second sphere is not as refined as the first sphere, consistent with the error representation and adjoint solution.

2.4.2 Validation: mesh convergence

The workshop guidelines prescribe the computation and interpretation of the computational results in a slightly different way than what is usual. In particular, the main difference is that the convergence order is no longer specified via an infinitesimal function of the minimum cell diameter h of a given mesh, as, for example, the number α is the usual relation with $err \leq Ch^{\alpha}$, but rather as a function of a redefined $h = N^{\frac{1}{d}}$ where N is the total number of degrees of freedom used for simulation and d is the geometrical dimension of the computational domain.

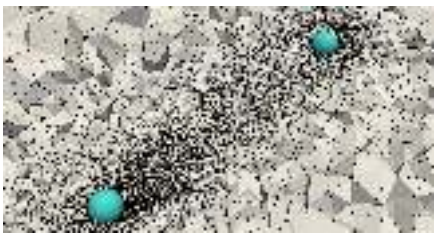
In our setting, this means that:

- Generalized length scale $h = N_{DOF}^{\frac{1}{d}}$
- Order of convergence $e(h) = Ch^p or log(e(h)) = plog(Ch)$
- Compute convergence sequence (e_i, h_i)
- Least-squares fit for p gives “effective order of convergence“.

Where, h is the minimum cell diameter, N is the number of degree of freedom and d is the dimension of computational domain.

Figure 2.12 shows the convergence of the drag coefficients of the first and second sphere by showing the approximation error as a function of the total number of degrees of freedom used in a particular simulation. Now, the one thing that certainly does not go unnoticed is that we have orders of convergence not only greater than one, as we were expecting, but also greater than two. This very curious effect

(a) FEniCS-HPC (DFS) refined mesh after 24 adaptive iterations (front view).



(b) FEniCS-HPC (DFS) refined mesh after 24 adaptive iterations (top view).

Figure 2.11

finds its roots in the way the mesh parameter h is defined as a metric of the made efforts. Indeed, the convergence order is now a measure of how good are the results we get are, as a function of the computational cost we paid to get them. Here, our adaptive refinement procedure comes into play, and it does so by playing a fundamental role. The effect of the adaptive procedure involves choosing the cells where more resolution is needed and refining them in order to reduce the error on the target cost functional. This is an equivalent formulation of the problem of finding the optimal way to spend a fixed amount of computational power in order to get the best possible approximation of our cost functional. The fact that we can get a good convergence order is proof of the fact that the method is indeed effective.

2.4.3 Validation: PyFR vs DFS

Figure 2.13 shows that mesh convergence drag C_d over adaptive mesh vertices. This also shows the mesh convergence is effective over more than 3. We also wanted to investigate how the mesh convergence for the given mesh and using PyFR would be. Figure 2.20 shows the DFS mesh convergence for the mesh 1 with 3-5 and PyFR with mesh 1 with P1-3 (with orders of 1-3). Even though the mesh is pre-refined, it does not converge for the DFS and is 10% away from the drag. Moreover, a drag reduction in the second sphere. However in PyFR, there is no clear drag (see Figure 2.17, 2.18 and 2.19) or mesh convergence, and moreover, PyFR has a restriction with time step $dt \approx 10^{/4}$. This leads to higher cost when compared to DFS, and this can be, for example, seen in table 2.1. Even with Nek5000 it is a similar behavior as PyFR. Figure 2.15 shows the Q-criterion for the Tandem sphere at various time step and Figure 2.14 shows the Q-criterion for the PyFR simulation for the Tandem sphere.

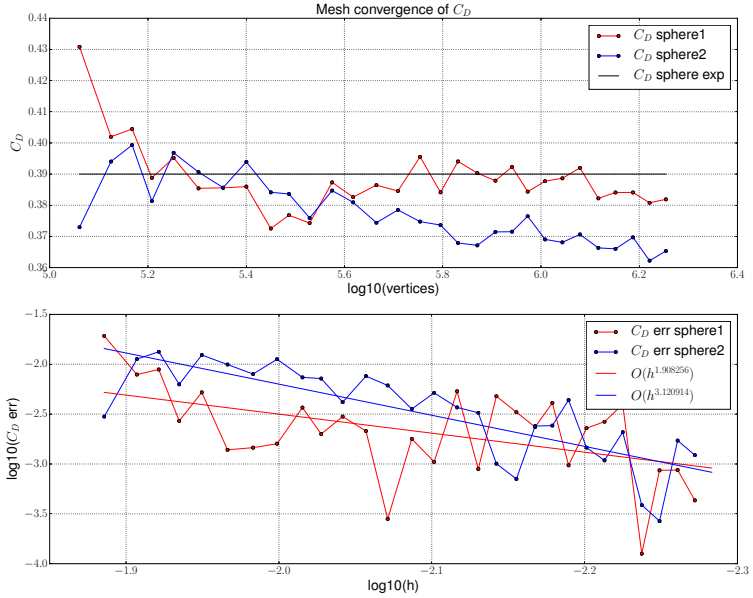


Figure 2.12: Time evolution of the drag coefficient for various iterations of our adaptive procedure.

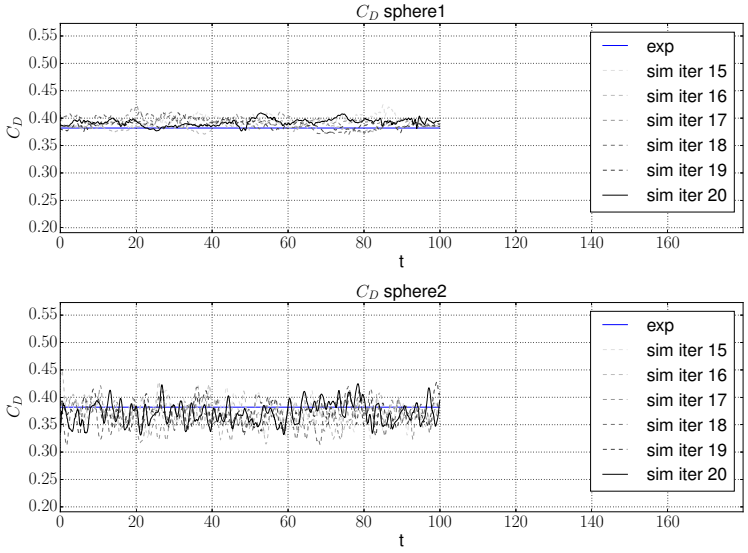


Figure 2.13: Mesh convergence of the drag coefficients of the two spheres.

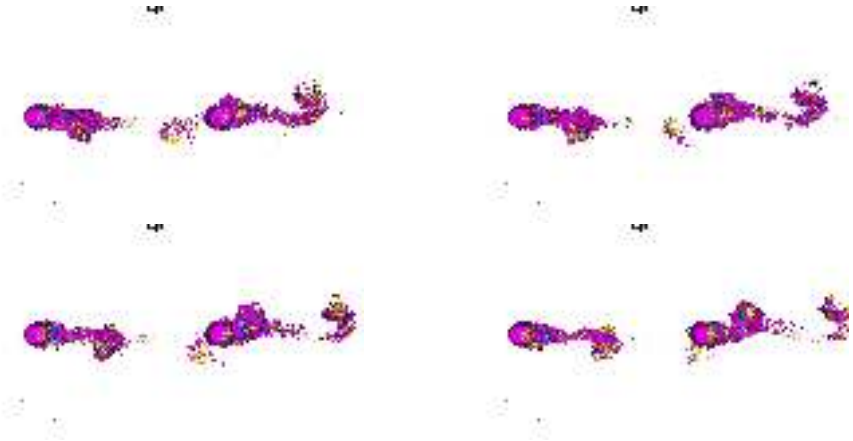


Figure 2.14: PyFR: Q-criterion of mesh refinement 0 with P2 with different time step.

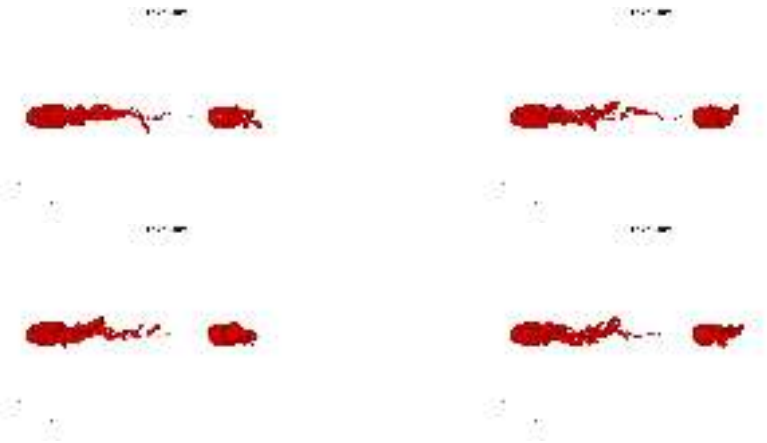


Figure 2.15: FEniCS-HPC: Q-criterion for the Tandem sphere with different timestep.

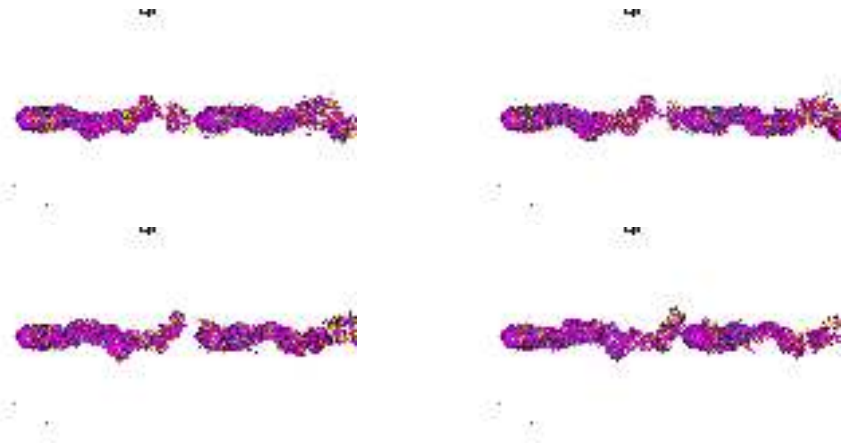


Figure 2.16: PyFR: Q-criterion of mesh refinement 1 with P2 with different time step.

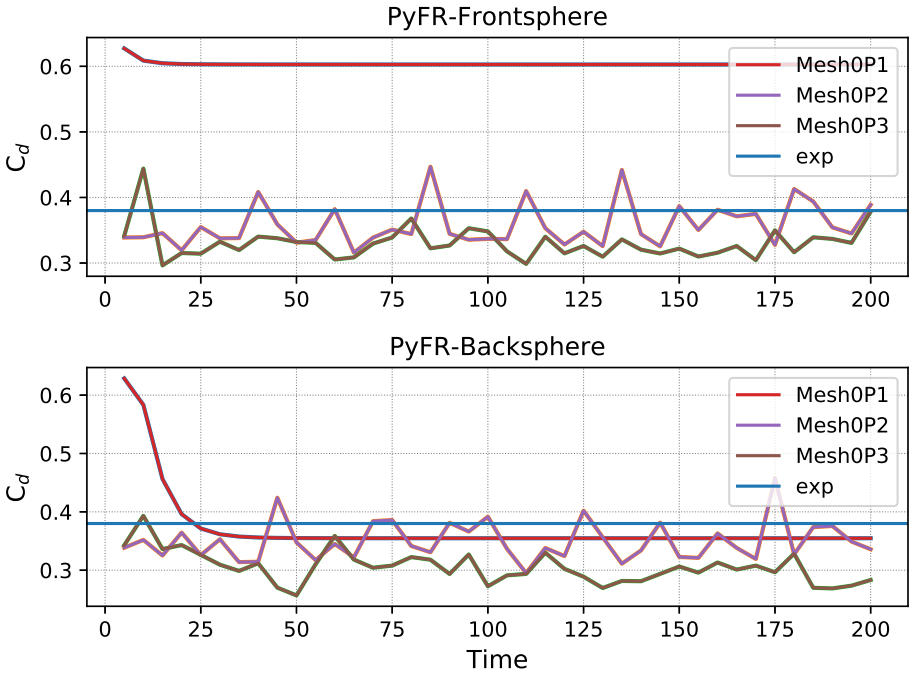


Figure 2.17: Mesh refinement 0 with different element orders for PyFR.

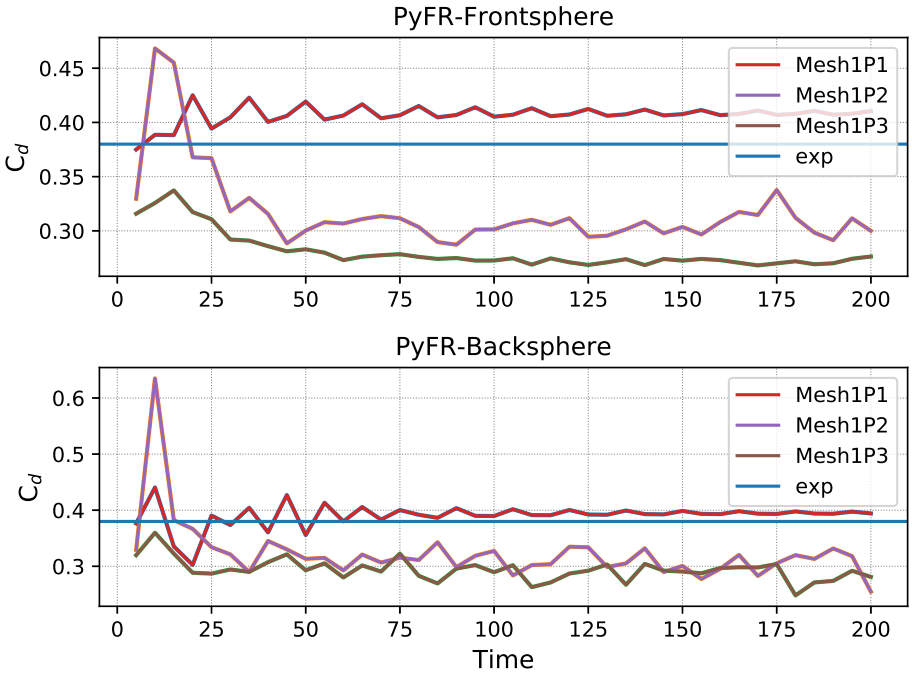


Figure 2.18: Mesh refinement 2 with different element orders for PyFR.

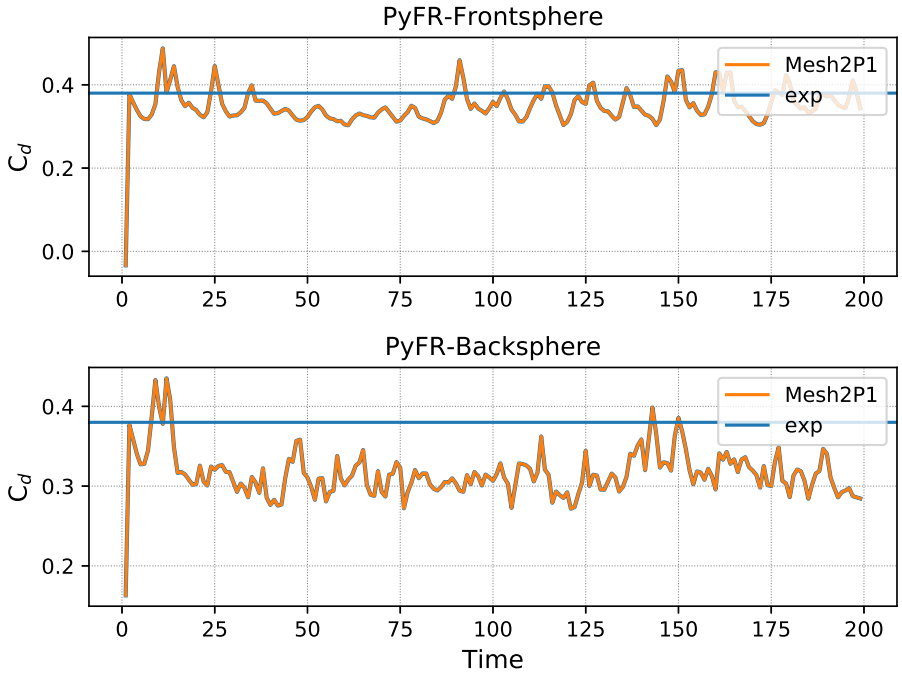
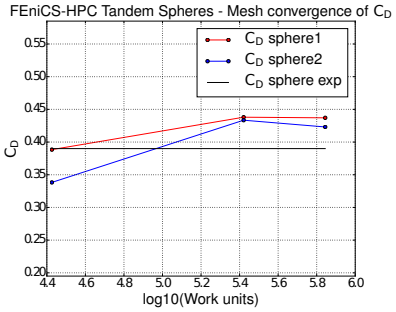
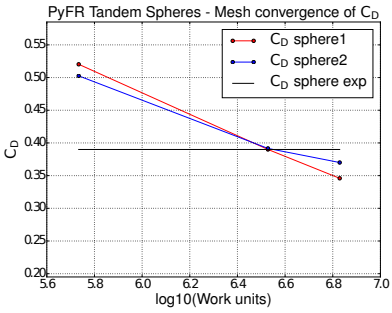


Figure 2.19: Mesh refinement 2 with different element orders for PyFR.



(a) FEniCS-HPC tandem sphere mesh for C_d convergence with provided mesh (for meshes 3,4 and 5).



(b) PyFR mesh convergence for C_d mesh 1 P1-3 (mesh refinement 1 with others 1-3).

Figure 2.20

Chapter 3

Applications

In this Chapter 3 we explain application related to predicting the aerodynamics forces for the relalistic aircraft, finding the velocity & pressure at the flash tank rain water storage tank and efficient design of 3D printing nozzle.

3.1 Predicting aerodynamics forces for the full aircraft with realistic Reynolds number

3.1.1 Background

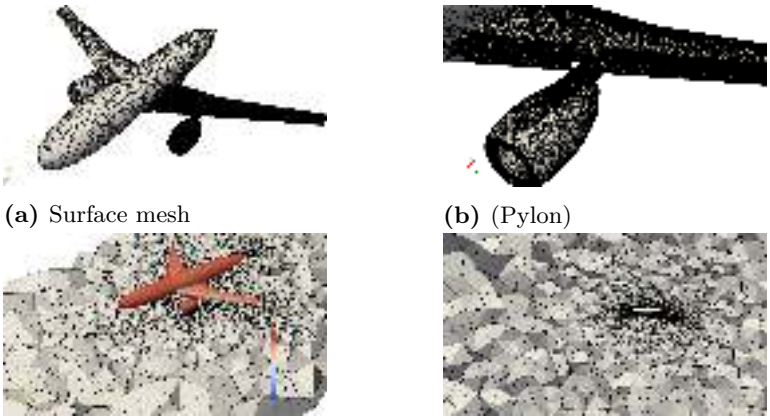
We present an adaptive finite element method for time-resolved simulation of aerodynamics without any turbulence model parameters, which is applied to a benchmark problem from the HiLiftPW-3 workshop on computing the flow past a JAXA Standard Model (JSM) aircraft model at realistic Reynolds number. The mesh is automatically constructed by the method as part of an adaptive algorithm based on a posteriori error estimation using adjoint techniques. No explicit turbulence model is used, and the effect of unresolved turbulent boundary layers are modeled by a simple parametrization of the wall shear stress in terms of a skin friction. In the case of very high Reynolds numbers, we approximate the small skin friction by zero skin friction, corresponding to a free slip boundary condition, which results in a computational model without any model parameter to be tuned, and without the need for costly boundary layer resolution. We introduce a numerical tripping noise term to act as a seed for the growth of perturbations, the results support that this triggers the correct physical separation at stall and has no significant effect pre-stall. We show that the methodology quantitatively and qualitatively captures the main features of the JSM experiment - aerodynamic forces and the stall mechanism - with a much coarser mesh resolution and lower computational cost than the state of the art methods in the field, with convergence under mesh refinement by the adaptive method. Thus, the simulation methodology appears to be a possible answer to the challenge of reliably predicting turbulent-separated flows for a

complete air vehicle.

3.1.2 Introduction

The main challenge today in Computational Fluid Dynamics (CFD) for aerodynamics is to reliably predict turbulent-separated flows [1,44], specifically for a complete air vehicle. This is our focus in this chapter.

We present an adaptive finite element method without turbulence modeling parameters for time-resolved simulation of aerodynamics, together with results stemming from the 3rd AIAA CFD High-Lift Prediction Workshop (HiLiftPW-3) which was held in Denver, Colorado, on June 3rd–4th 2017. The benchmark was a high-lift configuration of the JSM aircraft model shown in Figure 3.15 at a Reynolds number realistic for flight conditions.



(a) Surface mesh **(b)** (Pylon)

(c) Volume mesh **(d)** Wing slice

Figure 3.1: Overview of the JSM aircraft model and starting mesh for the adaptive method

The purpose of the workshop is to assess the capability of the state of the art CFD codes and methods.

Turbulent flows present features on a range of scales, from the scale of the aircraft down to the Kolmogorov dissipation scale. Direct numerical simulation (DNS) is not feasible for a full aircraft at realistic Reynolds numbers, instead the Reynolds Averaged Navier-Stokes equations (RANS) have long been the state of the art in industry [45]. RANS methods do not provide a full resolution of the flow field but simulate the mean field and introduce turbulence models to make it up for the unresolved dynamics. In particular, standard RANS do not resolve the transient flow field, but a statistical average of the turbulent flow.

In contrast, Large Eddy Simulations (LES) [46] resolve the dynamics of a filtered flow field, at the cost of higher mesh resolution than RANS, with subgrid models

for unresolved scales. Both RANS and LES, and hybrids such as DES, introduce model parameters that need to be tuned to the problem at hand, and the results are highly sensitive to the design of the computational mesh [47–51]. In particular, turbulent boundary layers cannot be resolved and must be modelled. Boundary layer models require tailored boundary layer meshes, which are expensive in terms of both mesh density and manual work. Witherden and Jameson in [44] state that "as a community, we are still far away from LES of a complete air vehicle".

The method we present is an adaptive finite element method without explicit turbulence model and boundary layer model, thus without model parameters and without the need for a boundary layer mesh. The mesh is automatically constructed by the method as part of the computation through an adaptive procedure based on a posteriori error estimation using adjoint techniques. Dissipation of turbulent kinetic energy is provided by residual-based numerical stabilization. The method is thus purely based on the Navier-Stokes equations, no other modeling assumptions are made.

We model the effect of turbulent boundary layers by a parametrization of the wall shear stress in terms of skin friction. For very high Reynolds numbers we approximate the small skin friction by zero skin friction, corresponding to a free slip boundary condition, which results in a computational method without any model parameters that need to be tuned, and without the need for costly boundary layer resolution.

In this chapter, we give the main components of the simulation methodology and we present our results stemming from the HiLiftPW-3, where we highlight the non-standard aspects of the methodology and discuss the results in relation to the experiments. The HiLiftPW-3 specified two variants of the JSM, one without pylon (or nacelle) and one with the pylon included in the geometry ("pylon on"). The difference in the aerodynamic forces between the two variants measured in experiments are small, typically less than 2 %. For this reason, we will focus only on the "pylon on" variant with the aim of validating our methodology.

The workshop guidelines prescribed the study of these two variants either with a fixed mesh or, more interestingly, using mesh adaptation techniques. Considering the nature of our method, which intimately depends on its adaptive procedure, we concentrated on the latter study. We did not use the provided computational meshes, but instead generated more suitable ones for our methodology, starting from the provided CAD files. We would like to point out that our adaptive methodology does not require any ad-hoc meshing procedure aimed at helping the solver identify flow features that are qualitatively known before starting the computations. Not only does this simplify the meshing procedure, which can now be carried out by non-specialized software (and scientists), but it also makes it faster: the only thing that we need is an initial mesh that captures the geometry of the object; this is due to the fact that the generated mesh loses memory of the underlying CAD model, and therefore the refinement of boundary triangles cannot correct a rough initial approximation of the CAD geometry. We plan to get rid of this constraint in the near future, implementing the functionality to refine boundary cells with the

new vertices projected on the CAD model. Once we have a sufficiently accurate surface description, however, we can let the mesh be coarse in the volume part, which will be refined iteratively by the adaptive algorithm.

This convenient approach allows us to perform computations starting with rather coarse meshes, increasing the number of cells only where needed in order to best utilize the available computational resources. Our initial mesh for the JSM case have about 2 5M cells.

We find that the simulation results compare very well with experimental data for all the angles of attack that we studied; moreover, we show mesh-convergence by the adaptive method, while using a relatively low number of spatial degrees of freedom. The low computational cost also allows for a time-resolved simulation, which provides additional results that cannot be obtained from a stationary simulation, such as the ones based on Reynolds-averaged Navier-Stokes equations (RANS).

Thus, the simulation methodology appears to be a possible answer to the challenge of reliably predicting turbulent-separated flows for a complete air vehicle. We specifically here present simulation results reproducing the physically correct stall mechanism of large-scale separation at the wing-body juncture, which is promising for our continuing work on validating the methodology.

3.1.3 Simulation methodology

In contrast to the statistical averages of RANS and the filtered solutions of LES, our simulation method is based on computational approximation of weak solutions to the Navier-Stokes equations (NSE), that satisfy the NSE in variational form integrated against a class of test functions.

Finite element methods (FEM) are based on a variational form of the NSE, and if the method satisfies certain conditions on stability and consistency, the FEM solutions converge towards a weak solution to the NSE as the finite element mesh is refined [9]. We refer to such FEM as a General Galerkin (G2) method or a Direct Finite Element simulation (DFS).

The resolution in DFS is set by the mesh size, and no turbulence model is introduced. The dissipation of turbulent kinetic energy in under-resolved parts of the flow is provided by the numerical stabilization of G2 in the form of a weighted least squares method based on the residual of NSE.

The mesh is adaptively constructed based on a posteriori estimation of the error in the chosen goal or target functionals, such as drag and lift forces. The a posteriori error estimates take the form of a residual weighted by the solution of an adjoint problem, which is computed separately using a similar stabilized FEM method [9]. The adaptive algorithm starts from a coarse mesh, which is locally refined each iteration based on the a posteriori error estimates.

We use a free slip boundary condition as a model for high Reynolds number turbulent boundary layers with small skin friction. This means that boundary layers are left unresolved and that no boundary layer mesh is needed.

This methodology has been validated on a number of standard benchmark problems in the literature [26–28, 52], including for an aircraft model for the HiLiftPW-2 [53] and we find that also for the benchmark considered in this chapter the method is very efficient and provides results close to the experimental reference data.

We have used a low order finite element discretization on unstructured tetrahedral meshes, which we refer to as cG(1)cG(1), id est, continuous piecewise linear approximation in space and time.

3.1.3.1 The cG(1)cG(1) method

Please refer to Chapter 2.1.1 for further information.

3.1.3.2 The Adaptive Algorithm

Please refer to Chapter 2.1.2 for further information.

3.1.3.3 A posteriori error estimate for cG(1)cG(1)

Please refer to Chapter 2.1.3 for futher information.

3.1.3.4 The Do-nothing Error Estimate and Indicator

Please refer to Chapter 2.1.4 for further information.

3.1.3.5 Turbulent boundary layers

In our work on high Reynolds number turbulent flows [29–31] we have chosen to apply a skin friction stress as wall layer model. That is, we append the NSE with the following boundary conditions:

$$\vec{u} \cdot \vec{n} = 0, \tag{3.1}$$

$$\beta \vec{u} \cdot \tau_k + \vec{n}^T \sigma \tau_k = 0, \quad k = 1, 2, \tag{3.2}$$

for $(\vec{x}, t) \in \Gamma_{solid} \times I$, with $\vec{n} = \vec{n}(\vec{x})$ an outward unit normal vector, and $\tau_k = \tau_k(\vec{x})$ orthogonal unit tangent vectors of the solid boundary Γ_{solid}. We use matrix notation with all vectors $\vec{v}$ being column vectors and the corresponding row vector being denoted by $\vec{v}^T$.

With skin friction boundary conditions, the rate of kinetic energy dissipation in cG(1)cG(1) has a contribution of the form

$$\sum_{k=1}^{2} \int_0^T \int_{\Gamma_{solid}} |\beta^{1/2} \bar{\vec{U}} \cdot \tau_k|^2 \, ds \, dt, \tag{3.3}$$

from the kinetic energy which is dissipated as friction in the boundary layer. For high Re, we model $Re \to \infty$ by $\beta \to 0$, so that the dissipative effect of the boundary

layer vanishes with large Re. In particular, we have found that a small β does not influence the solution [29]. For the present simulations we used the approximation $\beta = 0$, which can be expected to be a good approximation for real high-lift configurations, where Re is very high.

3.1.3.6 Numerical tripping

Please refer to Chapter 2.2.2 for further information.

3.1.3.7 The FEniCS-HPC finite element computational framework

The simulations in this article have been computed using the Unicorn solver in the FEniCS-HPC automated FEM software framework.

FEniCS-HPC [8] is an open-source framework for the automated solution of PDEs on massively parallel architectures, providing automated evaluation of variational forms whose description is given in a high-level mathematical notation, duality-based adaptive error control, implicit turbulence modeling by the use of stabilized FEM and strong linear scaling up to thousands of cores [10,11,18,54–57]. FEniCS-HPC is a branch of the FEniCS [12,13] framework focusing on high performance on massively parallel architectures.

Unicorn is solver technology (models, methods, algorithms and software) with the goal of automated high performance simulation of realistic continuum mechanics applications, such as drag or lift computation for fixed or flexible objects (FSI) in turbulent incompressible or compressible flow. The basis for Unicorn is Unified Continuum (UC) modeling [14] formulated in Euler (laboratory) coordinates, together with the General Galerkin (G2) adaptive stabilized finite element discretization described above.

The simulations in this chapter were run on supercomputer resources described in the Acknowledgments section, and took ca. 10h on the finest mesh for the whole time interval using ca. 1000 cores.

3.1.4 Results

We have performed simulations with the adaptive DFS methodology using the Unicorn/FEniCS-HPC framework for the JSM "pylon on" variant of the HiLiftPW-3 benchmark for the angles 4.36°, 10.58°, 18.58°, 21.57° and 22.58°. All angles except 22.58° have rich experimental data including forces, *cp* and oil film provided by the workshop, which we will compare against below. The angle 22.58° only has force data. The angles 21.57° and 22.58° exhibit stall in the experiment, e.g. large-scale separation leading to loss of lift force. Capturing stall quantitatively and with the correct stall mechanism is an open problem in aerodynamics, we therefore investigate both the angle 21.57°, which is the highest angle with detailed experimental data, as well as 22.58°,

The experiment is a semispan model at $Re = 1.93M$. However, "free air" computations were requested, and to avoid possible modeling errors introduced by a symmetry plane we model the entire aircraft. However, we choose the output quantity as drag and lift of the left side of the aircraft only, to save computational resources, where we expect the adaptive method to refine in the right half-volume only when there is a significant error contribution to the drag and lift on the left side.

The initial mesh in the adaptive method has ca. 2.5M vertices, and the mesh is then iteratively refined with 5% of the cells in every iteration until we observe mesh convergence in drag and lift, or as many times as we can afford. The finest adapted meshes in our computations presented here have 5M to 10M vertices.

We solve the time-dependent Navier-Stokes equations (2.1) with a non-dimensional unit inflow velocity over the time interval $I = [0, 10]$. For some of the cases close to stall where we observe a longer startup, we extend the time interval to $I = [0, 20]$. To compute the aerodynamic coefficients we take the mean value in the last quarter of the time interval, e.g. $[7.5, 10]$ or $[15, 20]$, respectively.

We have divided this section into three parts:

1. Detailed comparison of aerodynamic forces against the experiments including convergence of the adaptive method and analysis of stall.

2. Detailed comparison of the pressure coefficients cp against the experimental data, including analysis of cp in the stall regime.

3. Flow visualizations are presented, including dual quantities acting as weights in the error estimates, and comparison of surface velocity against oil film visualizations in the experiment.

3.1.5 Aerodynamic Forces

$$F = \frac{1}{|I|} \int_I \int_{\Gamma_a} p\vec{n}\, dsdt, \tag{3.4}$$

with Γ_a the left half-boundary of the aircraft. The drag and lift coefficients are then simply the x and y components of F since we have unit inflow.

We use the duality-based "do-nothing" adaptive method, which iteratively refines the mesh by repeatedly solving the primal and dual problem based on the a posteriori error estimate. This generates a sequence of adapted meshes, a procedure that takes the role of the classical *mesh study*.

In Figure 3.2 we plot the lift coefficient, C_l, and drag coefficient, C_d, versus the angle of attack, α, for the different meshes from the iterative adaptive method.

The size of the dots indicates the iteration number in the adaptive sequence, with larger dots indicating a larger number, that is more refinement. We connect the finest meshes with lines and also plot the experimental data as lines. For the angles

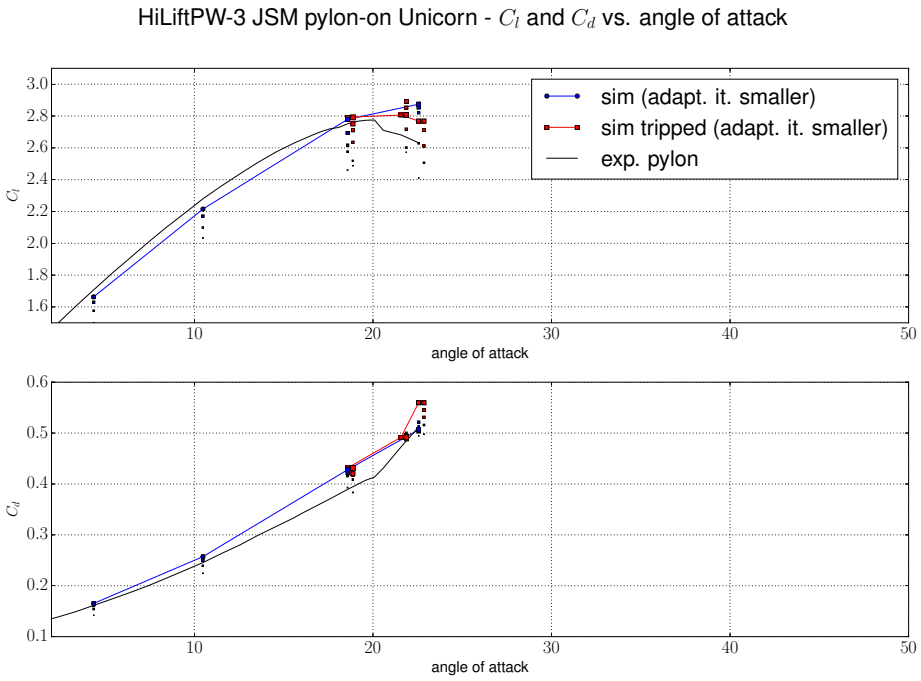


Figure 3.2: Lift coefficient, C_l, and drag coefficient, C_d, versus the angle of attack, α, for the different meshes from the iterative adaptive method.

18.58° and 22.58° we compute the solution both with and without the "numerical tripping" term described in Section tripping 3.1.3.6 to assess the dependence on the angle of attack, the tripped cases are plotted in red, and the adaptive sequence shifted somewhat to the right for clarity.

We observe mesh convergence to within 1 % to 2 % for all cases, a close match to the experiments for C_l, within circa 5 %, and a small overprediction of circa 10 % for C_d, which is consistent with the majority of the participants in HiLiftPW-3 across a range of methods [58], suggesting a systematic error in the problem statement or the experimental data.

For the stall regime angles 18.58°, 21.57° and 22.58° we qualitatively reproduce the stall phenomenon in the experiment – a decrease in C_l with increased angle of attack past 21.57°. We observe that the stall angle occurs somewhere between 18.58°, 21.57° which is ca. 1° from the experimental stall angle.

Additionally, we verify that the "numerical tripping" functions as expected: the term has no significant impact on the solution for an angle of 18.58°, which is the maximum lift angle and the maximum non-stalling angle, whereas for the

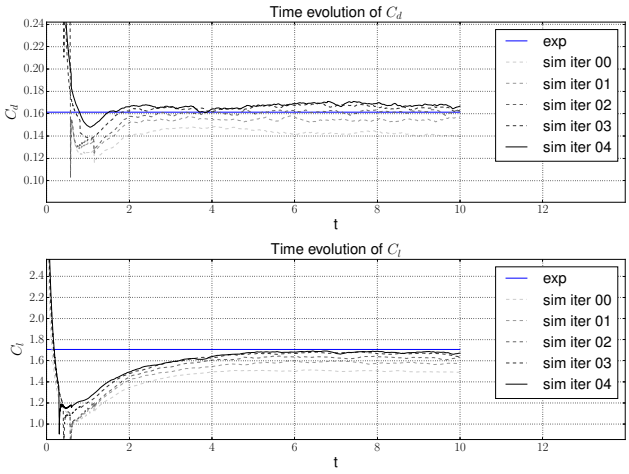


Figure 3.3: Time evolution of lift coefficient, C_l, and drag coefficient, C_d, and a table of the value for the finest adaptive mesh with relative error compared to the experimental results for $\alpha = 4.36\,^\circ$.

stalling angle 22.58° we observe that the tripping has the effect of triggering a large-scale separation consistent with the stall phenomenon, whereas the untripped case appears to contain too small perturbations for the separation to occur. We analyze the stall mechanism in more detail in the surface velocity visualization below.

To analyze the variability in time of C_d and C_l we plot the time evolution for $\alpha = 4.36°$ in Figure 3.3, untripped with $\alpha = 18.58°$ in Figure 3.4 and tripped with $\alpha = 18.58°$ in Figure 3.5.

For the pre-stall cases we observe an initial "startup phase" for $t \in [0, 5]$ and then an oscillation around a stable mean value. The effect of the numerical tripping is noise in the C_d and C_l signals with amplitude of about 1 %.

3.1.6 Pressure coefficients

The pressure coefficients C_p from both simulation on the finest adaptive mesh and experiments are plotted in Figures 3.7, 3.8 and 3.9, for the wing, flap and slat respectively.

The pressure sensor locations corresponding to the plots are specified in the diagram in Figure 3.6.

Since the aerodynamic force defined in (3.4) matches the experiment well, and since it consists of integrals of the pressure weighed by the normal vector, the C_p values also have to match the experiment on average. However, the C_p plots

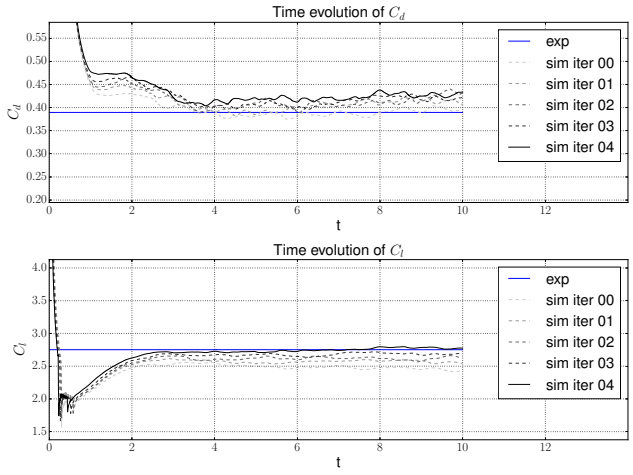


Figure 3.4: Time evolution of lift coefficient, C_l, and drag coefficient, C_d, and a table of the value for the finest adaptive mesh with relative error compared to the experimental results for $\alpha = 18.58\,^\circ$, untripped.

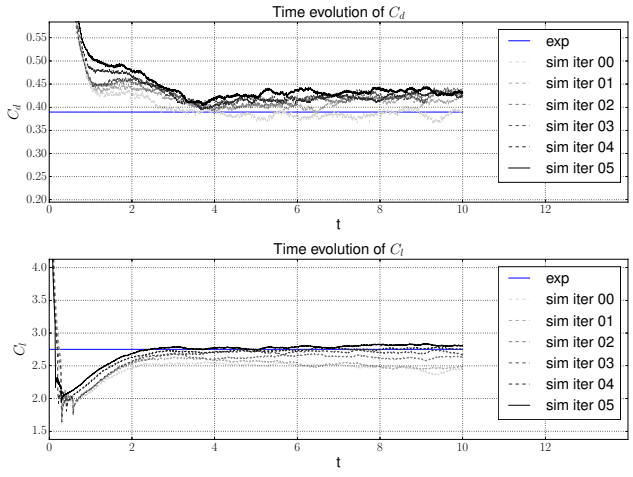


Figure 3.5: Time evolution of lift coefficient, C_l, and drag coefficient, C_d, and a table of the value for the finest adaptive mesh with relative error compared to the experimental for $\alpha = 18.58\,^\circ$ with numerical tripping.

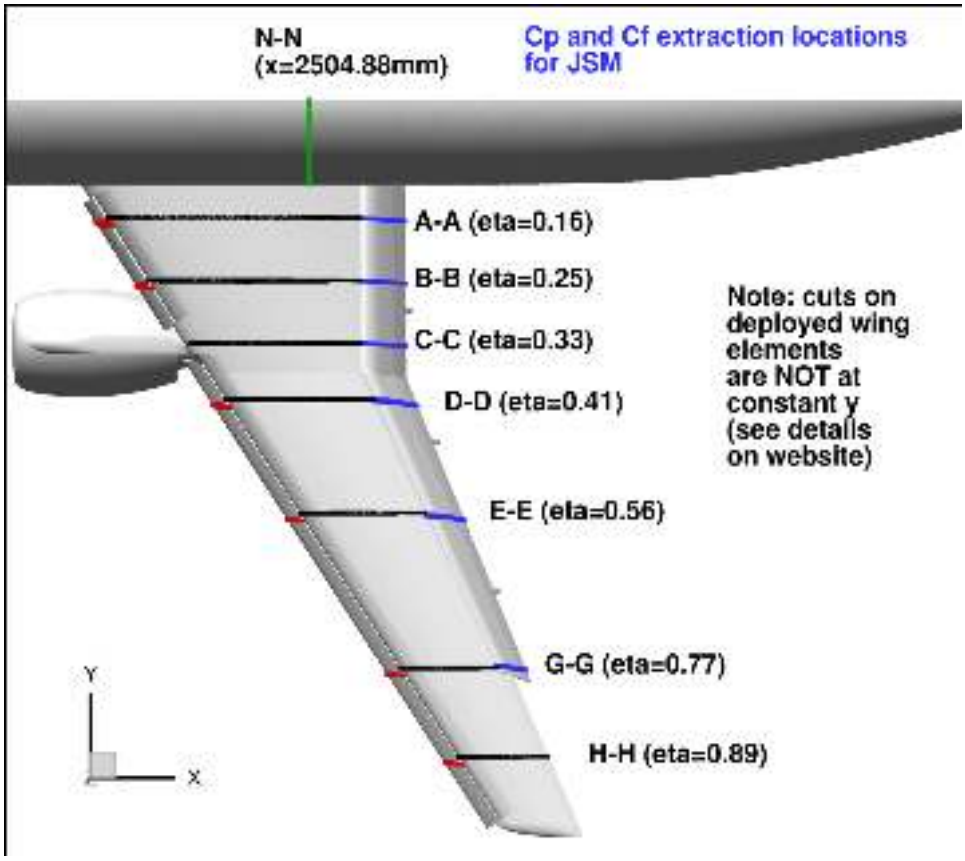


Figure 3.6: Diagram of the pressure sensor layout for the JSM configuration showing where the pressure sensors are located and how they are denoted.

can give insight into local mechanisms such as separation patterns, an important example being the stall mechanism. These local mechanisms are what we will focus on here.

First of all, we see that for the pre-stall angles $\alpha = 10.48\,^\circ$ and $\alpha = 18.58\,^\circ$ the simulation and experiment match very well for the wing and slat, and generally well for the flap, aside from local differences. The C_p for the simulation is lower on the upper surface for the flap close to the body (the A-A station). Otherwise, the curves generally match.

For the stall regime we analyze both $21.57\,^\circ$ where experimental C_p are available and $22.56\,^\circ$ where experimental C_p plots are not available. We compare both against the experimental C_p plots for $21.57\,^\circ$ to have a margin for if we have stall at a higher angle in the simulation. The simulation matches the experiment very well, there is a

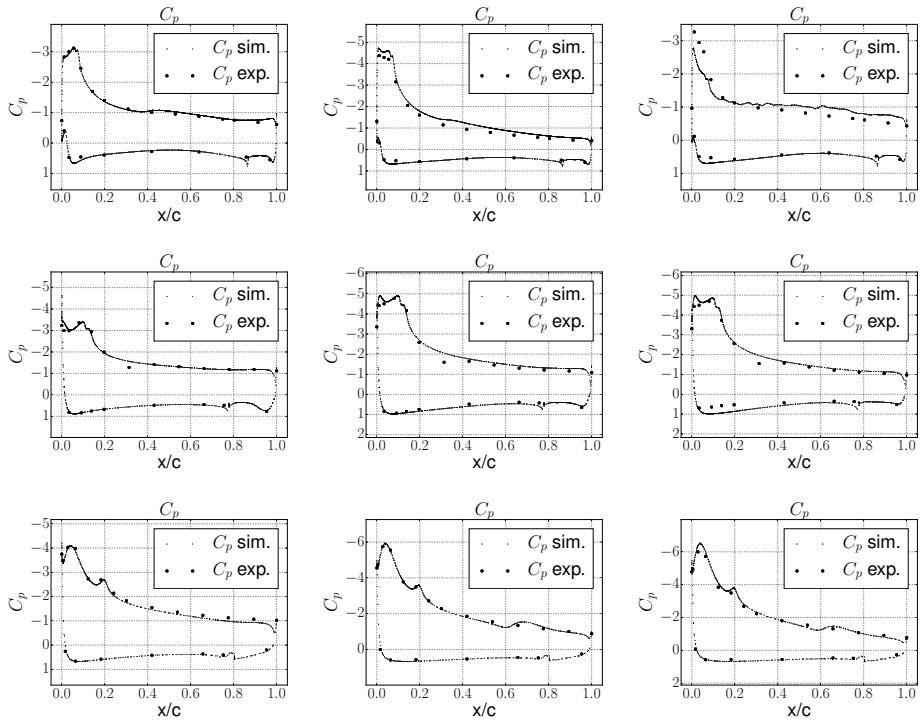


Figure 3.7: Pressure coefficients, C_p, versus normalized local chord, x/c, for the angles of attack $\alpha = 10.48\,^\circ$ (left), $\alpha = 18.58\,^\circ$ (middle) and $\alpha = 22.56\,^\circ$ (right) at locations A-A (top), D-D (middle) and G-G (bottom) for the wing of JSM pylon on.

small discrepancy for the wing close to the body (the A-A station), but considering that this is where the large-scale separation causing the stall is located, the results match acceptably.

The matching C_p curves are consistent with matching C_d and C_l from the aerodynamic force plots.

We now compare the tripped and untripped simulation with the experiment at $22.56\,^\circ$, as well as $22.56\,^\circ$ in Figure 3.10 for the wing.

We clearly see that the untripped simulation for $22.56\,^\circ$ grossly misses the C_p on the upper surface at station A-A, near the wing-body junction where the large-scale separation mechanism causing stall is located, while the tripped simulation captures the experimental C_p curve well, aside from a slightly lower C_p near the leading edge. We conclude that the tripping acts to trigger the physically correct separation. At the other stations, D-D and G-G, the tripped and untripped simulations are very similar, indicating that the tripping does not have a significant effect aside from

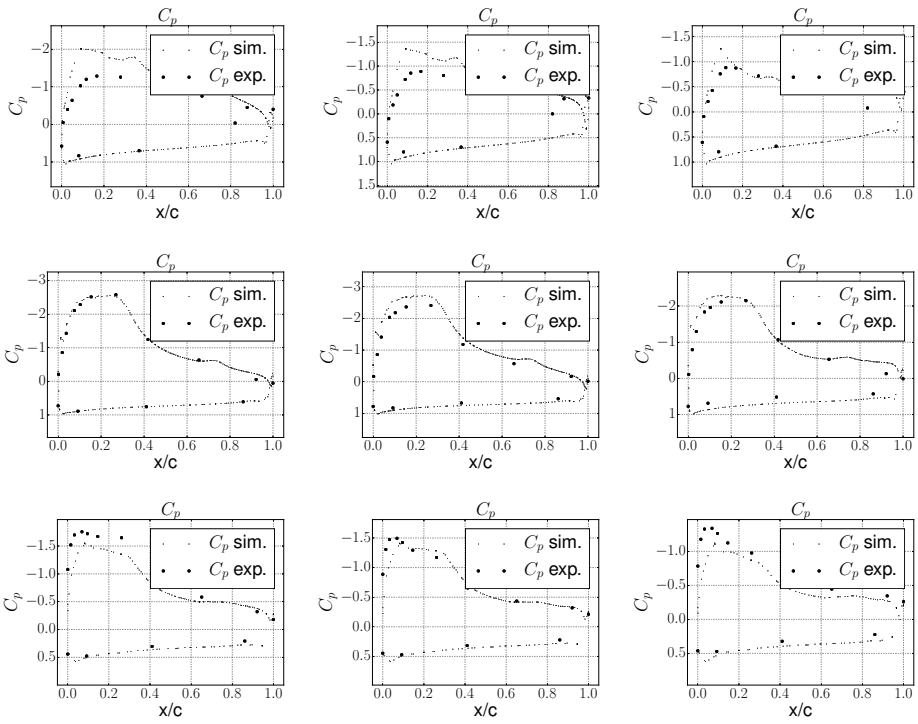


Figure 3.8: Pressure coefficients, C_p, versus normalized local chord, x/c, for the angles of attack $\alpha = 10.48\,^\circ$ (left), $\alpha = 18.58\,^\circ$ (middle) and $\alpha = 22.56\,^\circ$ (right) at locations A-A (top), D-D (middle) and G-G (bottom) for the flap of JSM pylon on.

the triggering of the perturbations.

The $\alpha = 21.57\,^\circ$ simulation is tripped and captures the experiment less well than $22.56\,^\circ$, but better than $22.56\,^\circ$ untripped indicating that we may have a ca. $1\,^\circ$ later stall angle in the simulation than in the experiment.

3.1.7 Flow and Adaptive Mesh Refinement Visualization

Here we concentrate on presenting effective visualization of the flow and the adaptive mesh refinement procedure. Our aim is to provide information on the properties and features of the approximated solution and, more importantly, of the approximating procedure, most of which cannot be discerned from one dimensional plots of the pressure coefficient and the aerodynamic forces. Sometimes these more complex visualizations cannot be directly compared to experiments, but still, they constitute a qualitative validation of the results.

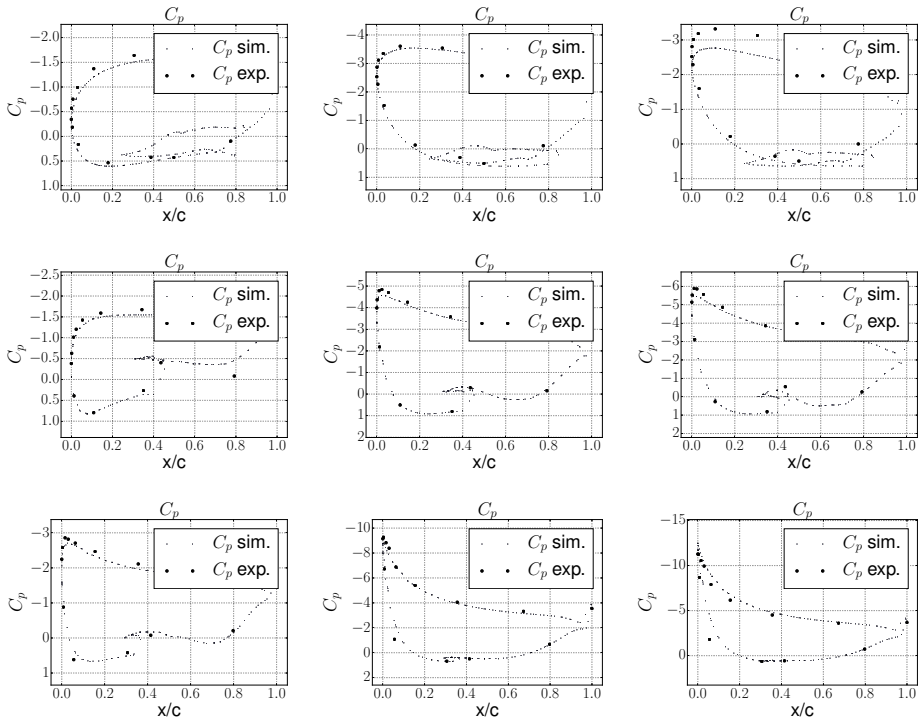


Figure 3.9: Pressure coefficients, C_p, versus normalized local chord, x/c, in the stall regime for the angles of attack $\alpha = 10.48\,^\circ$ (left), $\alpha = 18.58\,^\circ$ (middle) and $\alpha = 22.56\,^\circ$ (right) at locations A-A (top), D-D (middle) and G-G (bottom) for the slat of JSM pylon on.

The first plots that we show are the surface plots of the velocity magnitude on the upper side of the wing. Together with the velocity magnitude surface plots we also report pictures of the oil film experiment that was provided by the organizers as a validation tool. These serve as comparison tools, and we report such comparison in Figure 3.11.

Some common features intrinsic of the geometry of the JSM aircraft is revealed by the oil film experiment and reproduced by the velocity plots. A pattern of low velocity streaks, alternating with areas of higher velocity is seen on the suction side of the fixed wing for all angles of attack. This is caused by separation at the slat tracks upstream, which is correctly captured by the numerical solution.

Another characteristic feature of the flow is the turbulent separation near the tip of the wing. This is particularly evident in the case $\alpha = 18.59°$.

Areas that exhibit this kind of flow behavior influence the aerodynamic forces on

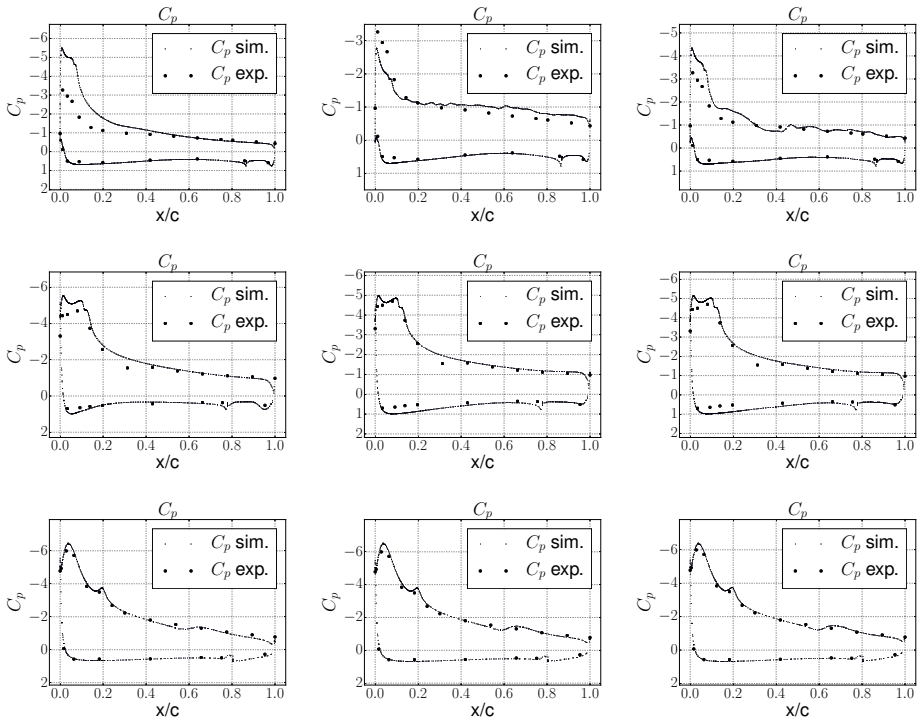


Figure 3.10: Pressure coefficients, C_p, versus normalized local chord, x/c, for the angle of attack $\alpha = 22.56\,^\circ$ untripped (left), the same angle $\alpha = 22.56\,^\circ$ tripped (middle) and $\alpha = 21.57\,^\circ$ tripped at locations A-A (top), D-D (middle) and G-G (bottom) for the wing of JSM pylon on.

the aircraft, and indeed in our experimentation we found that computations done on some meshes resulted in wrong predictions of the target functionals, usually yielding lower lift coefficients than the experimental ones. We were able to overcome this intermediate obstacle by refining the surface mesh was the original geometry had a higher curvature. We later interpreted the effectiveness of this workaround as a symptom that the original meshes were unable to capture the surface geometry to a sufficient degree of accuracy, and were for this reason failing at reproducing these complex patterns.

Another interesting visualization technique, which we are about to present is more closely related to turbulence itself: the Q-criterion [59]. The Q-criterion was widely used in the literature to visualize turbulent features of fluid flows. The main idea is that it is possible to define a quantity, commonly denoted by the letter Q, whose value is related to the vorticity and thus the visualization of the isocontours

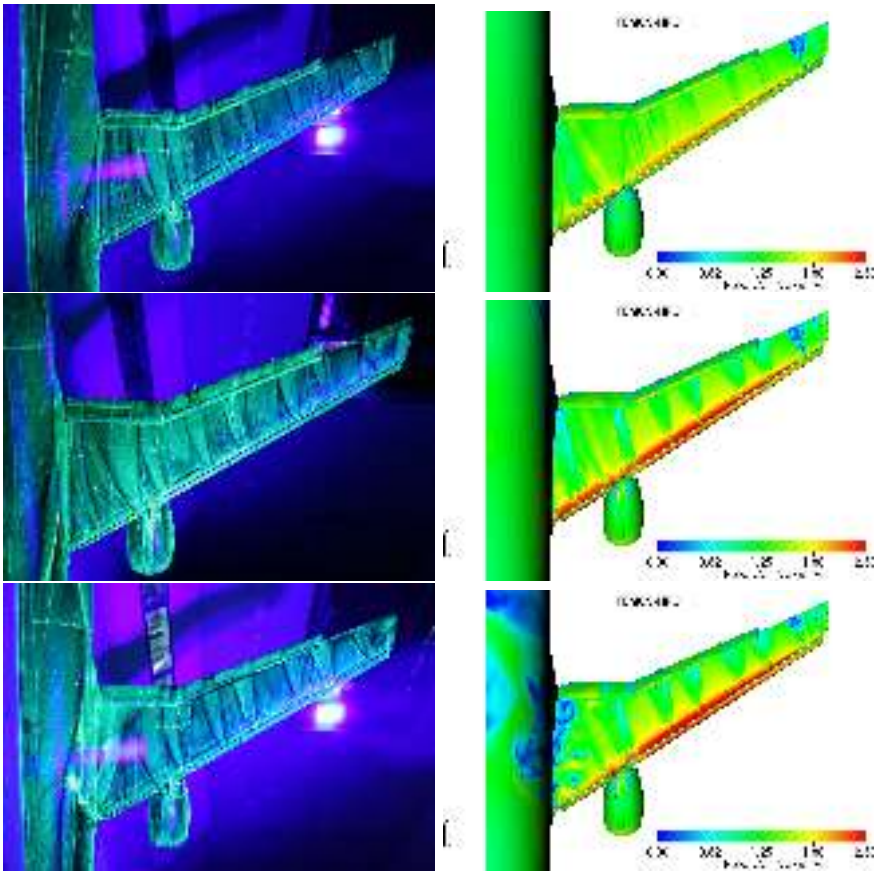


Figure 3.11: Comparison between experimental oil film visualization (left) and surface rendering of the velocity magnitude (right).

of Q is claimed to give visual information on the presence and location of vortexes within the flow field.

The Q-criterion for the case of the airplane with pylon is displayed in Figure 3.12 for three different angles of attack.

Once again, the visualization technique highlights the same pattern as in the previous case: the isosurfaces assume a characteristic V shape along the interfaces between the fast and slow velocity regions on the suction side of the wing. Not only that, but we can also clearly distinguish a clustering of these isosurfaces near the tip of the wing, matching the position of the turbulent separation zone that we mentioned above. The Q-criterion visualizations are consistent with the surface velocity plots, and this internal coherence increases our trust in the computational results.

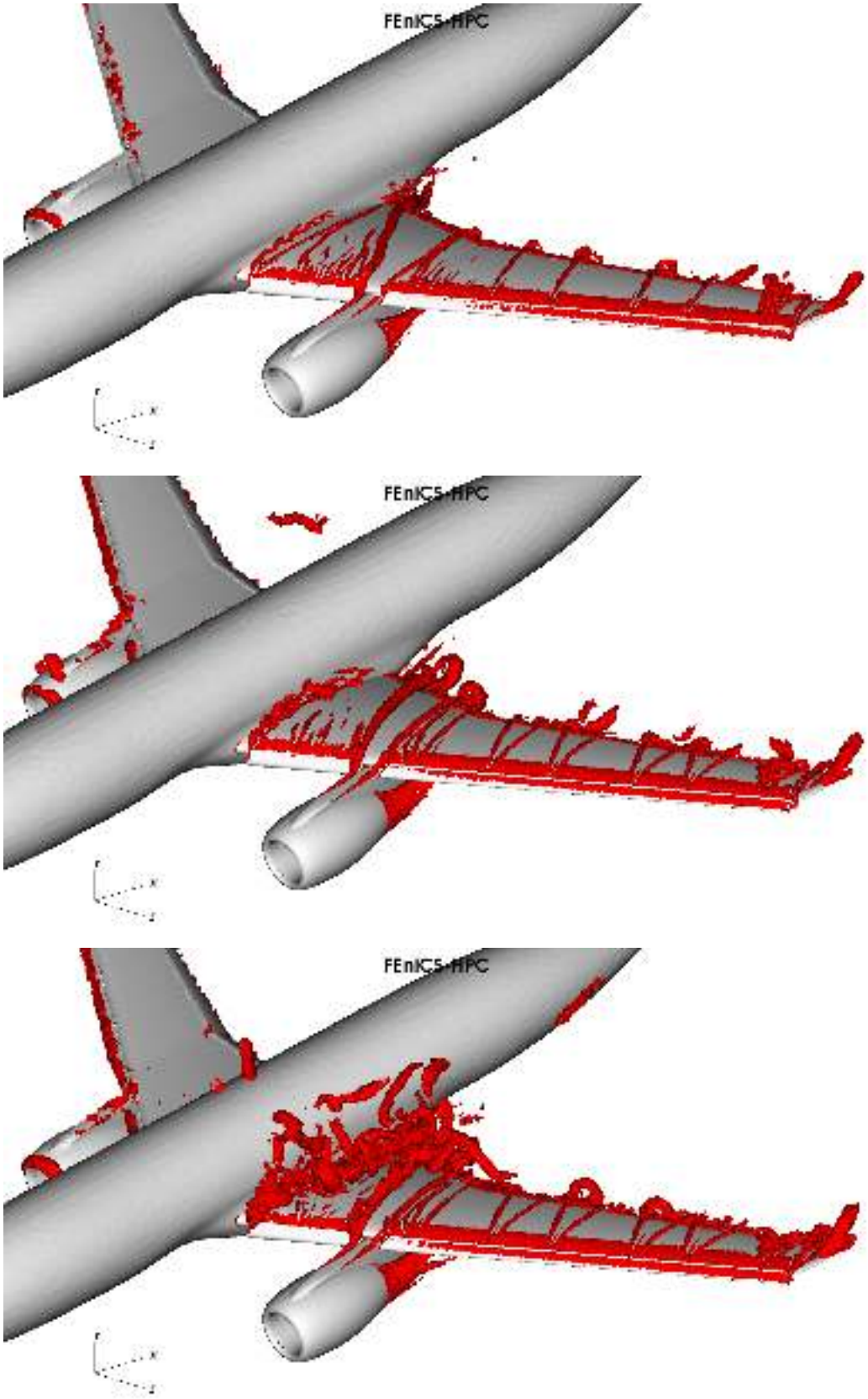


Figure 3.12: Instantaneous isosurface rendering at the final time of the Q-criterion with value $Q = 100$.

Let us now turn our attention to the adaptive procedure which produces the successive approximations of the fluid flow. As we described above, the mesh refinement solution is driven by the residual of the Navier-Stokes equations and the solution of the dual Navier-Stokes equations. We begin by showing a plot of a volume rendering of the dual solution, see Figure 3.13.

What is worth noting here is that the adjoint velocity flows backward in time and, consequently, it appears to be flowing in the opposite direction of the primal velocity. We observe that the part of the mesh where the dual velocity has higher values are *upstream* to the airplane. Because of the way the do-nothing error estimator is designed, we expect that the refinement will happen where both the residual and the dual solution are large. Indeed, this has the important implication that the mesh refinement will not only happen on the wing, where the forces are computed but also upstream, splitting cells that, a priori, are unrelated to the computation of the aerodynamic forces.

This feature is unique for our methodology: while other methods tend to refine the mesh in zones where *intuitively* higher accuracy would yield better approximation of the aerodynamic forces, namely around the body and downstream, the adaptive algorithm provides an automatic procedure that knows nothing about the features of the flow but only takes into account the residual of the equations of motion and the solution of the dual problem.

In our numerical experimentation we found that this is exactly what happens, as we are about to show. Consider Figure 3.14, showing a crinkled slice of the mesh for the initial and the finest meshes for a given angle of attack. It is clear that the mesh refinement procedure is concentrating both on the area around the surface where the aerodynamic forces are computed and in the upstream region. Some cells are refined downstream due to the large residual.

3.1.8 Conclusions

This chapter presents an adaptive finite element method without turbulence model parameters for time-dependent aerodynamics, and we validate the method by simulation results of a full aircraft model originating from the 3$^{\text{rd}}$ AIAA CFD High-Lift Prediction Workshop (HiLiftPW-3) which was held in Denver, Colorado, on June 3$^{\text{rd}}$-4$^{\text{th}}$ 2017. The mesh is automatically constructed by the method as part of the computation through duality-based a posterori error control and no explicit turbulence model is used. Dissipation of turbulent kinetic energy in under-resolved parts of the flow is provided by the numerical stabilization in the form of a weighted least squares method based on the residual of the NSE. Thus, the method is purely based on the NSE mathematical model and no other modeling assumptions are made.

The DFS method and these simulations are thus *parameter-free*, where no a priori knowledge of the flow is needed during the problem formulation stage, nor during the mesh generation process. Additionally, the computational cost is drastically reduced by modeling turbulent boundary layers in the form of a slip boundary condition, and thus no boundary layer mesh is needed.

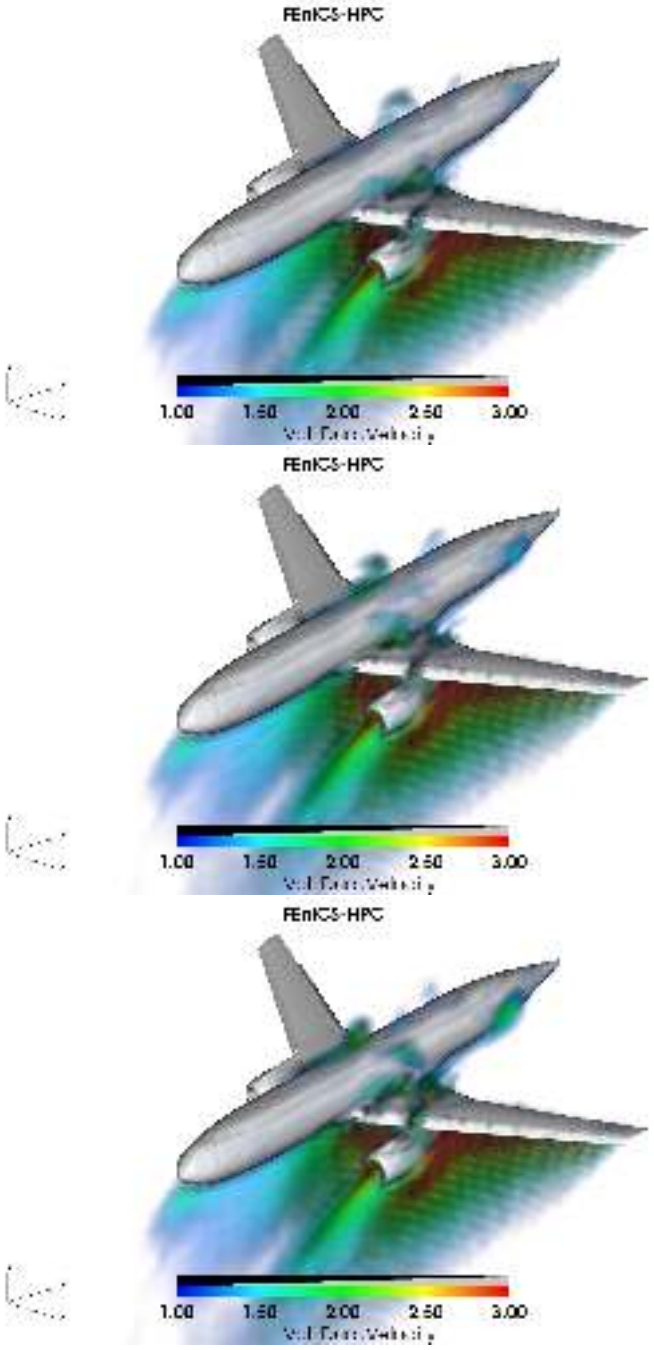


Figure 3.13: Volume rendering of the time evolution of the magnitude of the adjoint velocity $\vec{\varphi}$ magnitude, snapshots at t = (16, 18, 20).

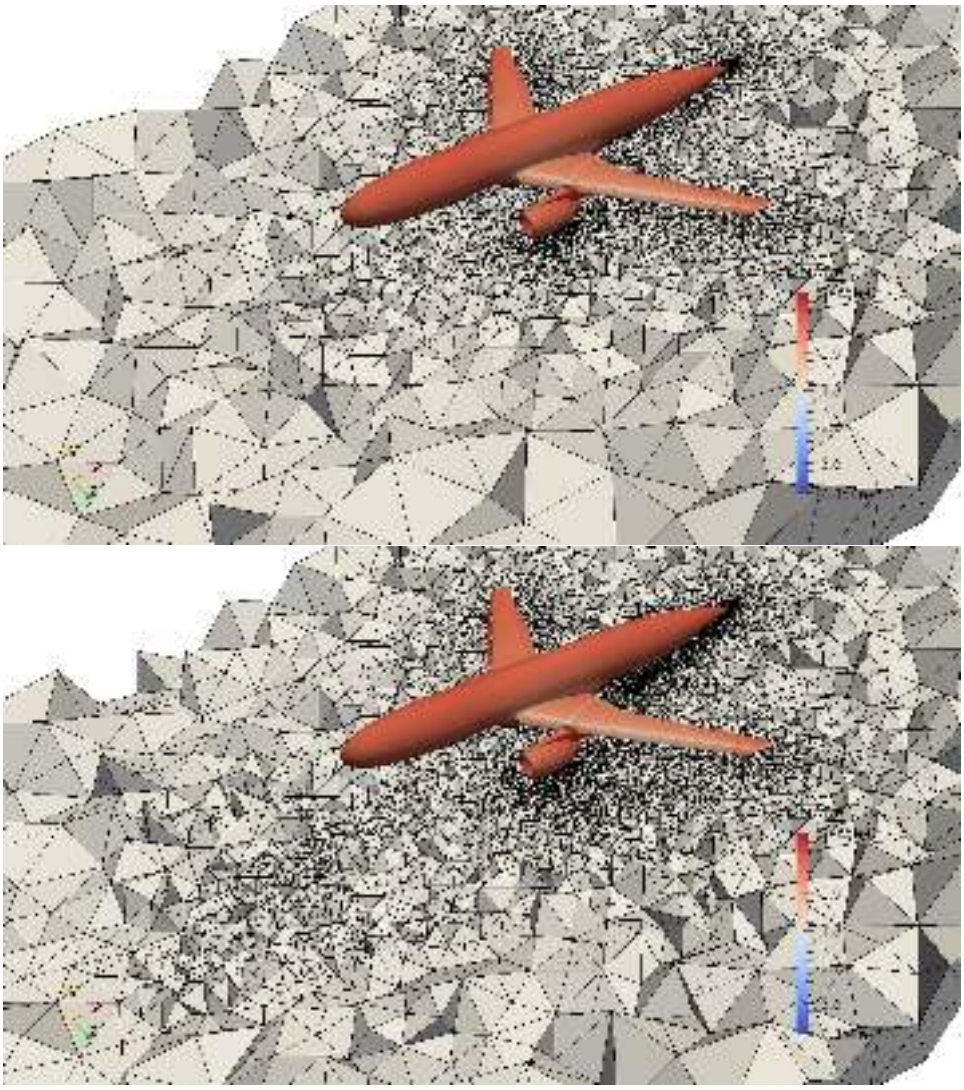


Figure 3.14: Crinkled slice aligned with the angle of attack, $\alpha = 10.48°$

The computed aerodynamic coefficients are very close to the experimental values for all the angles of attack that we studied. In particular, C_l is within circa 5 % of the experiments, C_d has a small overprediction of circa 10 %, which is consistent with the majority of the participants in HiLiftPW-3 across a range of methods [58], suggesting a systematic error in the problem statement or the experimental data.

The fact that the error is automatically estimated by the method is itself a critical feature missing in most (if not all) other computational frameworks for CFD.

Moreover, the adaptive procedure in DFS is seen to converge to a mean value with oscillations of the order of 1 % to 2 %. This contributes to increase the confidence in the numerical method.

The point of adaptive computations is all about saving on the computational cost. During the workshop, we had the chance to compare our performance with that of the other participating groups. In terms of the number of degrees of freedom, DFS is about ten times cheaper than the leading RANS and Lattice Boltzmann Methods.

To capture stall, we applied a tripping noise term that turned out to have the effect of triggering the physically correct stall separation pattern. A similar idea with a noise term is employed in the DNS community as well, and the addition of this term seems to have no effect on non-stalling configurations, which is an important validation.

We observed that DFS was able to capture the stall mechanism of the proposed configuration, namely the large scale separation pattern that occurs at the wing-body juncture. The same mechanism is observed in the experiments. The stall angle is also captured within ca. 1 °.

3.2 Turbulent Multiphase Flow in Urban Water Systems and Marine Energy

High-Reynolds number turbulent incompressible multiphase flow represents a large class of engineering problems of key relevance to society. Here we describe our work on The Consorcio de Aguas Bilbao Bizkaia is constructing a new storm tank system with an automatic cleaning system, based on periodically flushing tank water out in a tunnel. Here we study the MARIN benchmark modeling breaking waves over objects in marine environments. Both of these problems are modeled in the Direct FEM/General Galerkin methodology for turbulent incompressible variable-density flow [60, 61].

3.2.1 Overview

The present world is facing global warming by anthropogenic activities and natural disasters (e.g. earthquake, tsunami, and volcano), which are causing ozone layer depletion, loss of biodiversity, water quality, and climate change. Among these,

climate change causes extreme weather phenomena across the world, resulting in either severe flood or drought. In 20th century, about 1000000 people were killed and 1.4 billion people were affected by the floods [62]. Floods not only kills and affects the people and also it affects the eco-system, agriculture production, infrastructure and creates economical instability. According to The International Disaster Database [63], from January 1975 to June 2002, flash floods (due to heavy rain) in Europe has 5.6 % morality (rate of killed verses affected people) [62]. Especially, in Spain from 1900 to 2016, flash flood killed 987 people, 1350 people were affected and it caused damage of 642000000 US$ [63]. And it is predicted that the weather instability (more flash floods) going to be happen frequently in coming years [64].

Bilbao is located northern part of Spain, and it has Oceanic/Atlantic climate; its annual precipitation is from 1200 to 2000 millimeter (mm) [65]. Consorcio de Aguas Bilbao Bizkaia (BWC) is constructing a new storm tank (detention tank) system with an automatic cleaning system based on a periodically flushing tank out in a tunnel in Galindo. This would prevent rainwater goes into a river and also minimize the hydraulic load on the existing sewer infrastructure. Later this water can be treated in a wastewater treatment plant (WWTP) in Galindo for portable or other purposes.

The excessive water from the detention tank overflows through the tunnel, where the sediments and floating objects might permanently settle down on the surface of the tunnel, which will eventually give the foul smell, and in a frequent run, it will also affect the downstream flow in the tunnel. It is not a feasible solution to send men to clean the tunnel. Instead, BWC wants to clean the tunnel with periodic flushing using water from the detention tank.

Our work in the research is to predict the velocity, pressure and flow rate in the down stream side of the tunnel. Where this velocity, pressure and flow rate values will be used as an input parameter for the shallow water modelling; and also, velocity at the door section will be used to design the stronger gate to open a water from the detention tank. We used the Finite Element Method (FEM) for the simulation calculation; the problem is modelled as a 3D computations of the primitive equations (variable-density incompressible Navier-Stokes) in FEniCS-HPC.

For the simulation, we have investigated the 4 options, they are: T_D=5s,10s and H=6m,10m, with T_D the time for the door to fully open and H the initial water height in the tank. We compute the time interval I=[0s, 6s] for T_D=5s, and I=[0s, 11s] for T_D=10s.

3.2.2 Mathematical modelling

Please refer to the Chapter 2 for the multiphase mathematical modelling.

3.2.3 The Bilbao Water Consortium storm drain problem

The problem consists of an initial water volume stored in a tank with a gate opening toward a storm drain tunnel. To clean the tunnel, the gate is opened, and water flows out throughout the tunnel. Here we investigate a range of gate opening speeds and initial water height in the tank.

The geometry of the tank, door, and start of the tunnel is presented in Figure 3.15. The mesh is refined close to the door of the tank, and in the region with x coordinate [30m, 40m] at the start of the tunnel, giving ca. 800k mesh points. The door height is 1m and breadth is 6m. A slice of the mesh is presented in Figure 3.17 demonstrating the distribution of the cell size.

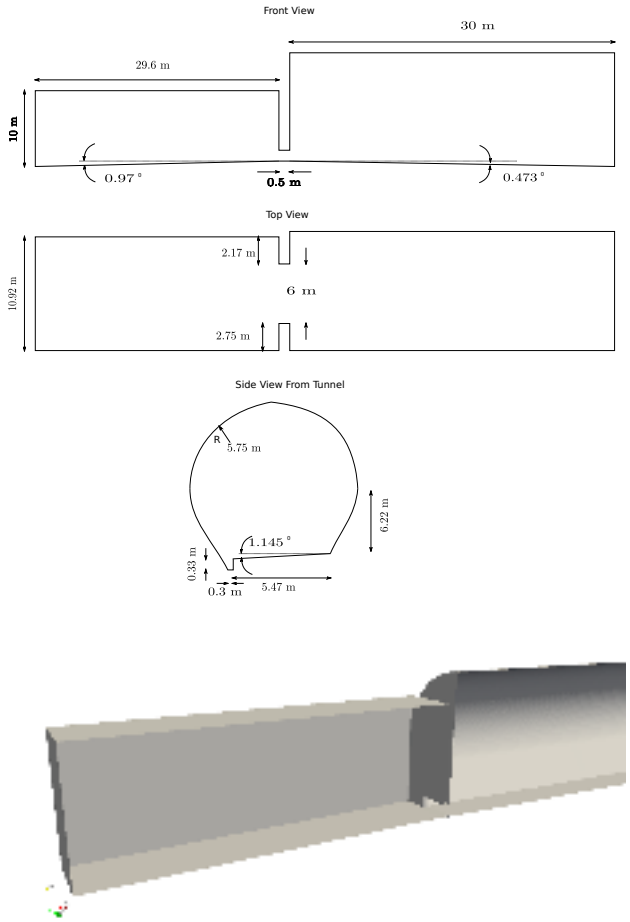


Figure 3.15: Schematic of the geometry of the tank, door and start of the tunnel (top), and a 3D rendering (bottom).

Gravitational force g=9.81 m s^{-1} is set at top of the tank and P=0 Pa is set at the end of the tunnel. Walls in the geometry at the upstream & downstream side considered as a free-slip boundary condition. The rest of the space in the geometry (tank and tunnel) were set to air density at ambient temperature as well as water density is set to the water in the tank. We set time interval as 0.02s, for example, in T_D=5s, we get 250 time interval samples. The door opening mechanism is based on the time interval (0.02s). In this case, for each time sample, the door is moving (flow region space is gradually increasing) 0.004m (total length for gate opening is 1m) towards the upward vertical direction. Figure 3.16 shows the boundary and

initial conditions set up in the simulation.

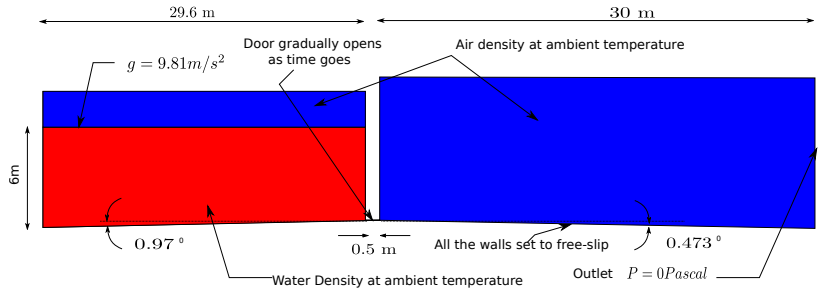


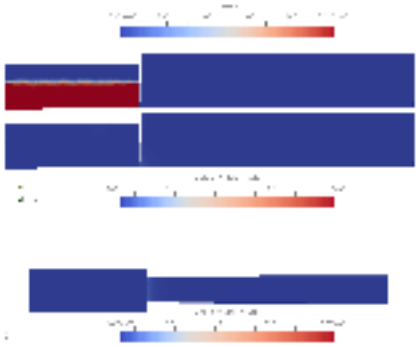
Figure 3.16: Initial and boundary conditions set up.

Figure 3.17: Slice plot through the x-z plane (front view) of the mesh.

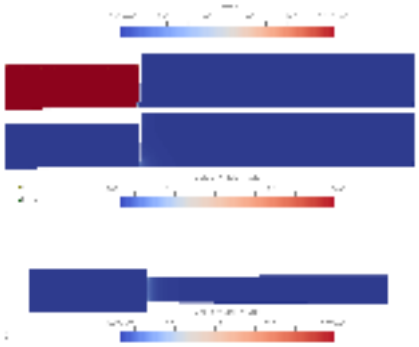
3.2.4 Simulation results

The simulations are carried out by running on the Beskow supercomputer at KTH (Sweden). The output is a piecewise linear density field representing the density of air and water, and a velocity field for the entire continuum. An average time step of ca. $k = 5 \times 10^{-4}$ is chosen, giving ca. 10k time steps for T_D=5s and 20k time steps for T_D=10s. We used 1024 cores on the Beskow Cray XC40 system at KTH for each simulation, giving ca. 1s per time step of computation time.

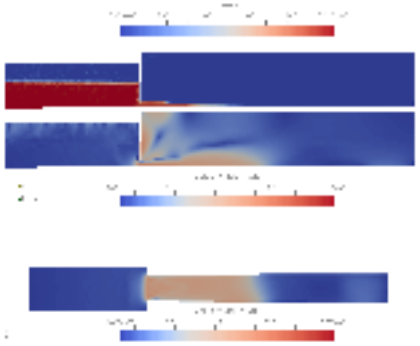
In this section, we plot slice plots of the density (showing the evolution of the water surface), the velocity, 3D plots of the isovolume of the density (showing the evolution of the water surface). Additionally, we plot the flow rate through the door over time, and the average velocity in the door section and in the first 10m section of the tunnel.



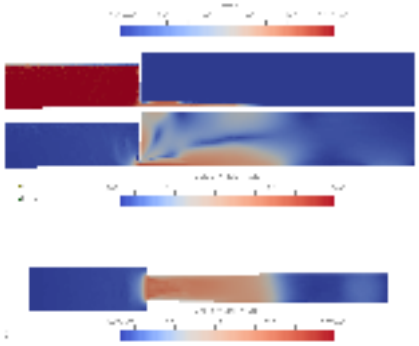
(a) Density and velocity x-y and x-z slice T_D={5s,10s}, H=6m, t=0.



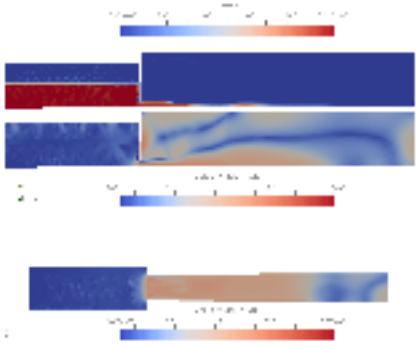
(d) Density and velocity x-y and x-z slice T_D={5s,10s}, H=10m, t=0.



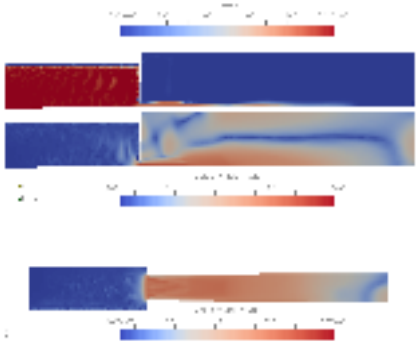
(b) Density and velocity x-y and x-z slice T_D=5s, H=6m, t=5s.



(e) Density and velocity x-y and x-z slice T_D=5s, H=10m, t=5s.



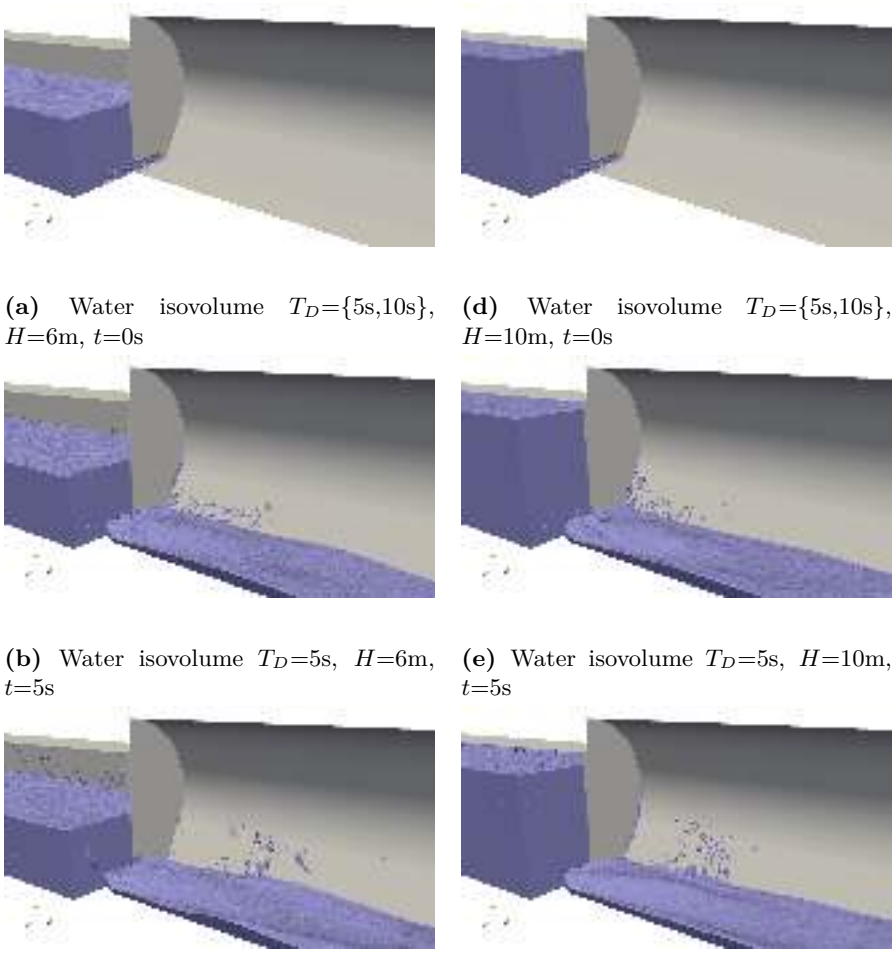
(c) Density and velocity x-y and x-z slice T_D=10s, H=6m, t=10s.



(f) Density and velocity x-y and x-z slice T_D=10s, H=10m, t=10s.

Figure 3.18: Density and velocity x-y and x-z at different height with different door opening time.

For example, Figure 3.18 shows the density and velocity at different time and height setting. As we can see in Figure 3.18 (b) and (e), the turbulence behaviour is slightly higher for Figure 3.18 (e); as well as the velocity at the bottom of the tunnel is also higher in both cases (T_D={5s and 10s}) for H=10m.



(a) Water isovolume T_D={5s,10s}, H=6m, t=0s

(d) Water isovolume T_D={5s,10s}, H=10m, t=0s

(b) Water isovolume T_D=5s, H=6m, t=5s

(e) Water isovolume T_D=5s, H=10m, t=5s

(c) Water isovolume T_D=10s, H=6m, t=10s

(f) Water isovolume T_D=10s, H=10m, t=10s

Figure 3.19: Water isovolume at different height with different door opening time.

Figure 3.20 shows the flow rate through the gate. As we can see here, gate opening time and water height do have an influence over the water flow rate through the opening gate. The volumetric flow rate depends on the area and the velocity. For example, in case H=10m, when T_D=10s, the area of the flow through the

gate is increasing slowly compare to T_D=5s, this shows, T_D=5s reaches volumetric flow rate $50\,\mathrm{m}^3\,\mathrm{s}^{-1}$ at 5th second, whereas T_D=10s reaches volumetric flow rate $50\,\mathrm{m}^3\,\mathrm{s}^{-1}$ at 10th second. In general, if the H is same (for 5s and 10s) then the flow rate depends on the area of the opening gate and if the T_D is the same (for 6m and 10m) then flow rate depends on the velocity.

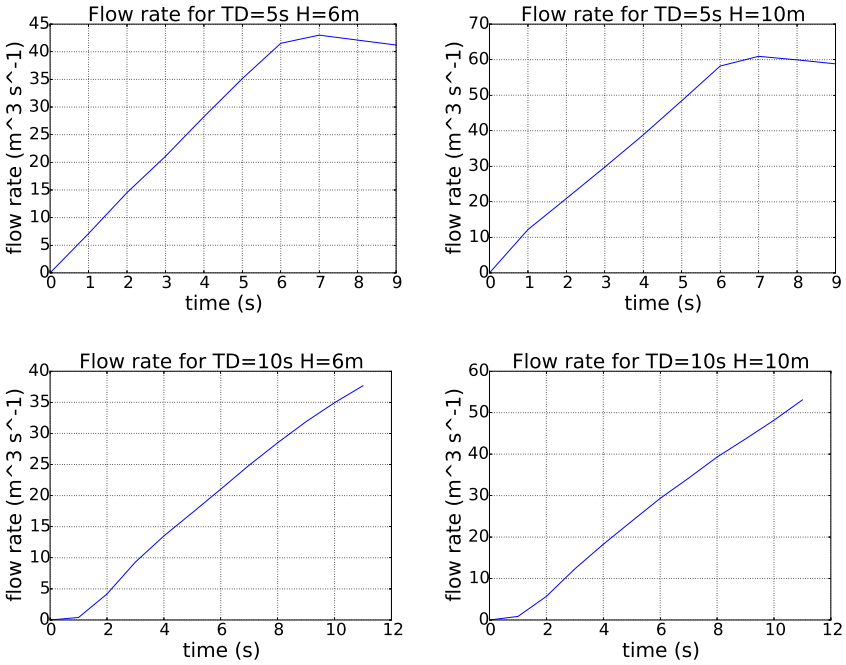


Figure 3.20: "Spending" flow rate through the door.

Figure 3.21 shows the average velocity at the door section. It is clearly seen that, velocity will be higher at the door section for H=10m compare to H=6m (for the same opening time T_D=5s). From Pascal's Law, we can calculate the static pressure at the gate, but in the real case as the gate opens we need to calculate the dynamic pressure, from this we can calculate the force acting on the door section.

For example, static pressure can be expressed as $P = \rho g h$Pa, considering H=10m, we get P=98.100Pa. Similarly, the dynamic pressure at a fully opened door can be calculated as $P = \frac{1}{2}\rho v^2$Pa. The velocity is around $10\,\mathrm{m\,s}^{-1}$ when the door is fully opened for T_D=5s and H=10m, from this the dynamic pressure will be P=490.500Pa. A force can be calculated from $F = pa$ N. The area of the door opening is $6\,\mathrm{m}^2$ and P=490.500Pa, then the maximum acting force at the door will be F=2943 N.

The average velocity at the downstream side of the tunnel is plotted in Figure 3.22. It shows, for example, when T_D=5s and H=6m, the velocity is increasing gradually up to $6\,\mathrm{m\,s^{-1}}$ until the time reaches 4s, after that the velocity is started to reduce. But on the other hand, if the T_D=6s and H=10m, it reaches higher velocity upto about $6\,\mathrm{m\,s^{-1}}$. This behaviour is similar to if T_D=10s, thus concludes, water height influences average velocity at the bottom of the tunnel.

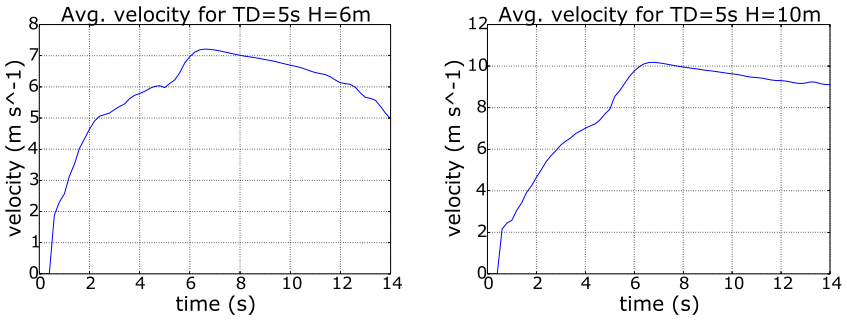


Figure 3.21: Average x-velocity in the door section.

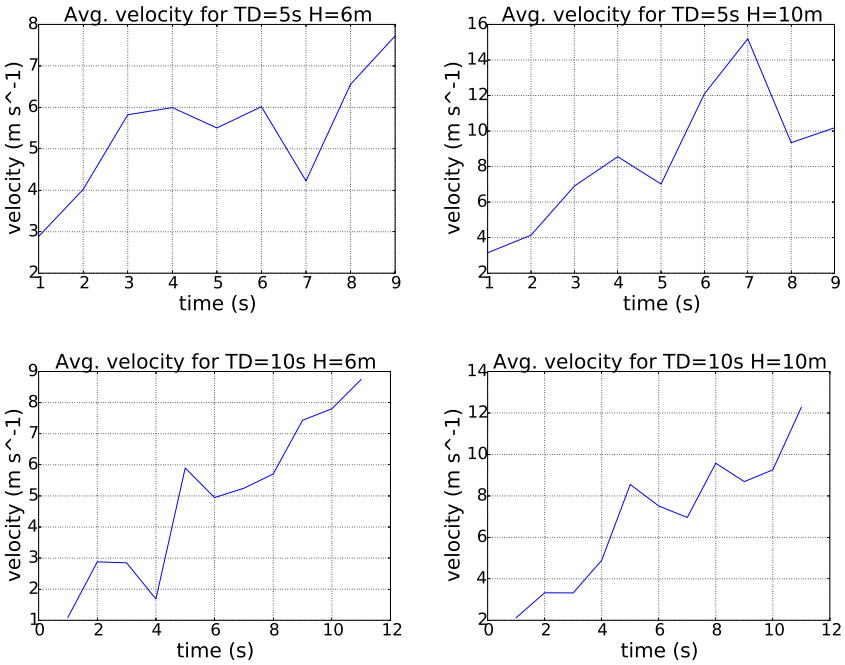


Figure 3.22: Average flushing x-velocity in the first 10m-section of the tunnel.

3.2.5 Conclusions

In this report, we provide computational results for Direct FEM simulations of the primitive 3D variable-density incompressible Navier-Stokes equations. The density and velocity fields have a 3D structure, a triangular jet shape, at the exit of the door. The door opening time does not appear to have a a significant influence on the structure or magnitude of the velocity. The water height in the tank has a significant influence on the magnitude of the velocity in the flushing section at the beginning of the tunnel.

3.3 3D printing Nozzle design

In this section, we present a nozzle design of the 3D printing using FEniCS-HPC as mathematical and simulation tool. In recent years 3D printing or Additive Manufacturing (AM) has become an emerging technology and it has been already in use for many industries. 3D printing considered as a sustainable production or eco-friendly production, where one can minimize the wastage of the material during production. Many industries are replacing their traditional parts or product

manufacturing into optimized or smart 3D printing technology. In order to have 3D printing to be efficient, this should have an optimized nozzle design. Here we design the nozzle for the titanium material. Since it is a metal during the process, it has to be preserved by the inert gas. All this makes this problem comes under the multiphase flow. FEniCS-HPC is high level mathematical tool, where one can easily modify mathematical equations according to physics and has good scalability on massively super computer architecture. And this problem modelled as Direct FEM/General Galerkin methodology for turbulent incompressible variable-density flow in FEniCS-HPC.

3.3.1 Objective

The overall goal of the FRACTAL project led by Etxe-Tar is to design a 3D printing nozzle for a selective laser melting method, where a fiber laser will be used as an energy source to melt an inter gas and powder mixture jet ejected by the nozzle. Where the entire metal melting process is confined by the inert gas (argon) to ensure minimizing oxygen interaction and hydrogen pick up. 3D printing, also know as additive manufacturing (AM), has gained popularity in recent years, especially in the medicine industries, where to make orthopedic components such as the knee, hip, jaw replacements [66, 67]; and also it uses increases in consumer products and mechanical industries. For example, General Electronics (GE) produces a 3D printing spare parts for its next generation LEAP jet engines [68]. And in medicine (bio-mechanical), each and every patient has a unique structure, to replace their body parts in a quick way, 3D printing is a good option. It is estimated that to produce a knee implant component with a traditional method produces up to 80% metal waste chips [69].

In order to design efficient 3D printing nozzle, we have conducted 3 stages of research for the nozzle, they are:

- Initial design
- Optimized design
- Compare the simulation results with experimental results

A efficient 3D printing nozzle should have this properties, which are as follows:

1. Minimize a wastage of the titanium powder (titanium is expensive)
2. Avoid oxidation during a melting process (might decrease the melting efficiency, nitrogen and oxygen pickup)
3. Minimize heating of tip of a nozzle (during the melting temperature might rise around 1,668 °C)

We consider a continuum multiphase model of the three phases, they are:

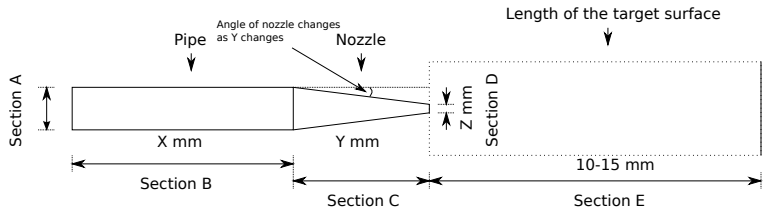


Figure 3.23: Schematic 3D printing nozzle design.

- Inert gas ans particle mixture
- Inert gas
- Air

In the presented simulations we omit the air phase for simplicity, but the model has the capability for including this third phase without significant extra complexity.The model is discretized by the Direct FEM Simulation (DFS) methodology in the FEniCS-HPC framework.

3.3.2 Mathematical modelling

Please refer to the Chapter 2 for the multiphase mathematical modelling.

3.3.3 Initial Design

First, we would like to see how the jet of flow will be look like in reality and how far it can be steady before it breaks; to do this, we have come up with a simple cone shape model. Figure 3.23 shows the initial design of the 3D printing prototype. FEniCS-HPC does not have adaptivity for the multiphase flow; in this case, we ran a couple of adaptive simulations for one-phase flow, and we took that mesh as an initial mesh for the multiphase flow, for example, this mesh can be seen in Figure 3.27. During the design phase the following items should be considered, they are, the laser beam diameter is 150 µm, and distance from a nozzle tip to the target surface should be between 10 mm to 15 mm

Figure 3.24 and 3.25 show a multiphase flow with velocities profiles and different section of cone size. As we can see in here, higher velocity seems to be stable compare to the lower velocity.

3.3.4 Optimized design

In this design phase we introduce a sheath flow [70], which will make the flow steady and narrow down a jet flow, this concept of geometry can seen in Figure 3.26. Sheath flow has a real benefit which can be seen in the Figures 3.28 and 3.29

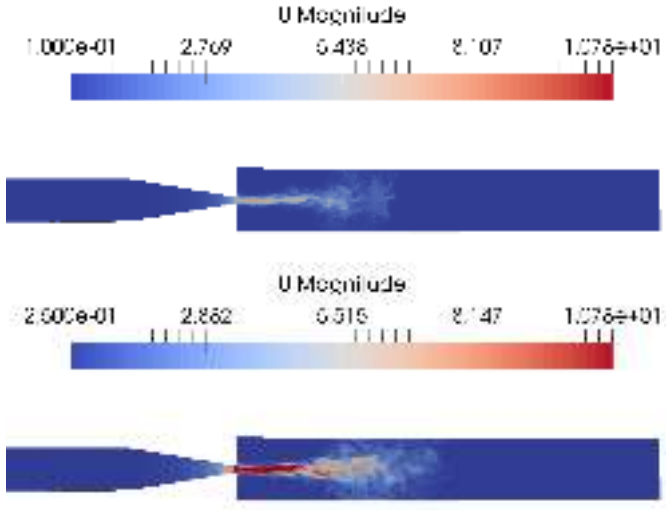


Figure 3.24: Nozzle length (section c)is 2.5mm and velocities ={0.1, 0.25} m/s

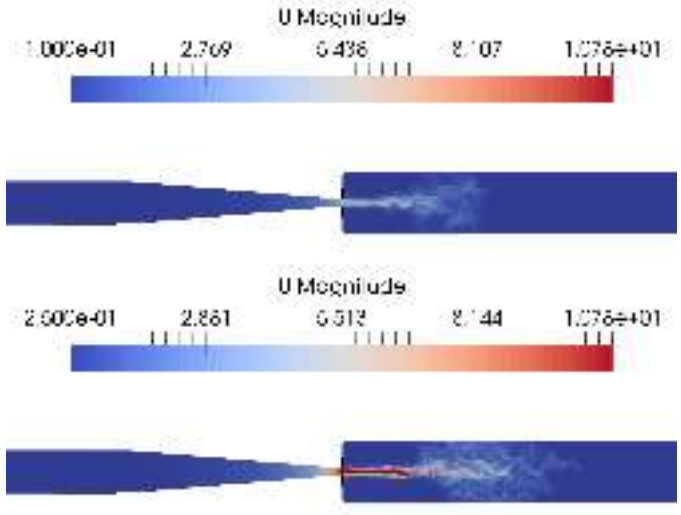


Figure 3.25: Nozzle length (section c)is 5.0mm and velocities ={0.1, 0.25} m/s

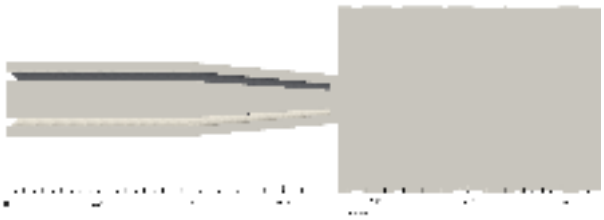


Figure 3.26: Schematic 3D printing sheath model

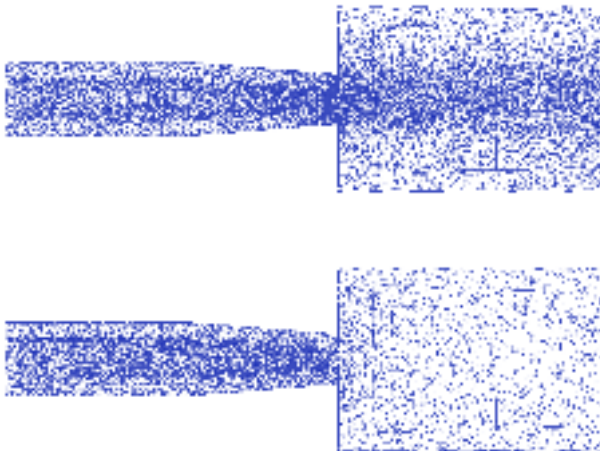


Figure 3.27: Adaptivity mesh for the single phase flow

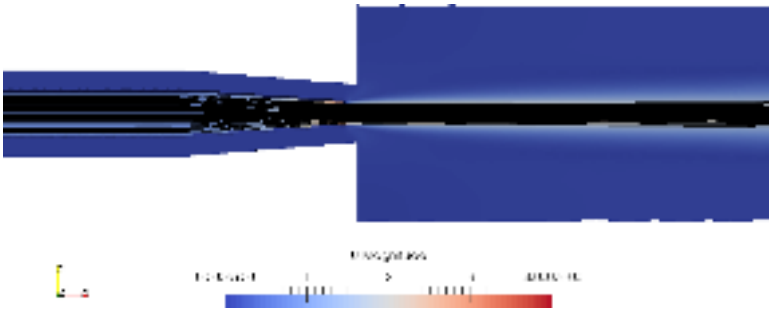


Figure 3.28: Schematic 3D printing sheath model

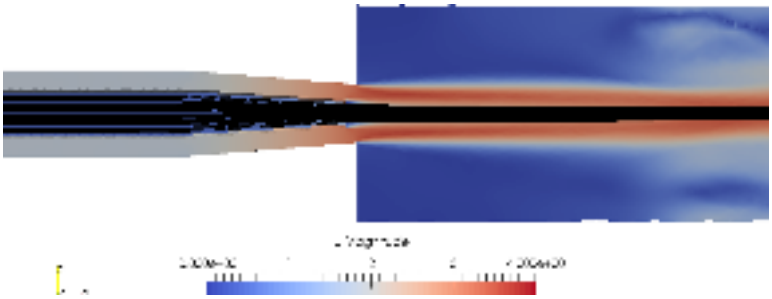


Figure 3.29: Adaptivity mesh for the single phase flow

3.3.5 Validation

In this stage, we got experimental results 3D printing nozzle, which is almost similar to the sheath modeling, which we discussed above. Figure 3.30 shows the design of the model and reference sample points location.

3.3.6 Results

The equation (2.3.1) can be scaled arbitrarily keeping the Reynolds number fixed, using the formula for the Reynolds number $Re = \frac{\rho \bar{u} L}{\nu}$ with $\bar{u}$ the freestream velocity, L the characteristic length and ν the viscosity.

In the presented simulations, we choose the physical geometrical dimensions, where L can be chosen as the diameter of the inner channel, $L = 0.8mm$. We choose $\rho_{mixture} = 1$, and $\rho_{inert} = 1e-3$. The inner inflow is chosen as $u_{inner} = 0.75$. We then study a range of sheath inflow velocities and viscosities to study the different flow regimes and the focusing effect of the sheath flow.

We give a schematic of plot lines in Figure 3.30, used for studying the density distribution in subsequent plots. The density field in a slice through the center of the domain is given in Figures 3.31, 3.32, 3.33 for a range of sheath inflow speeds indicated in the plots. In Figures 3.34, 3.35, and 3.36 the density along the specified plot lines.

We use the same mesh for all the simulations, which has been constructed by adaptive one-phase simulations, where we make the coarse approximation that the velocity field for one-phase flow will be similar to the multi-phase case in the present simulations.

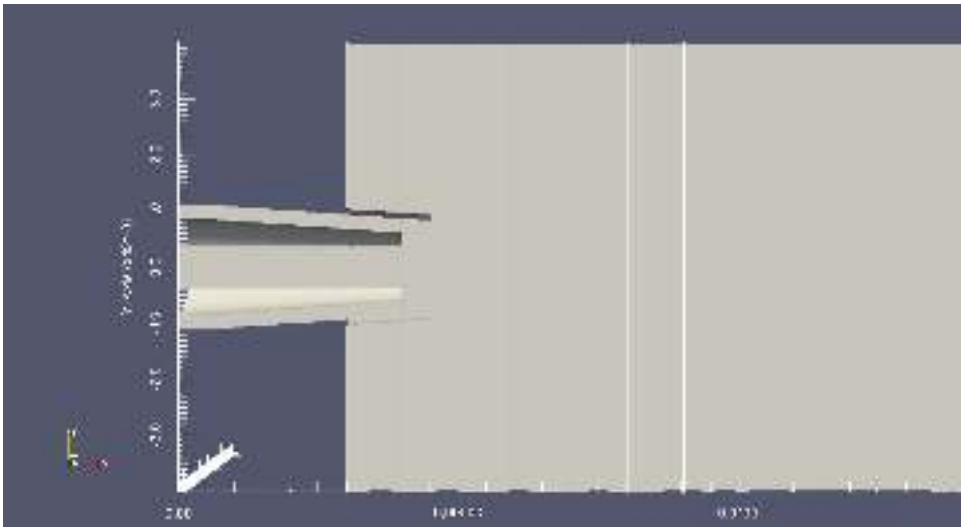


Figure 3.30: Plot line positions = 0.0, 1.0, 2.0, 3.0, 4.0 and 5.0 mm

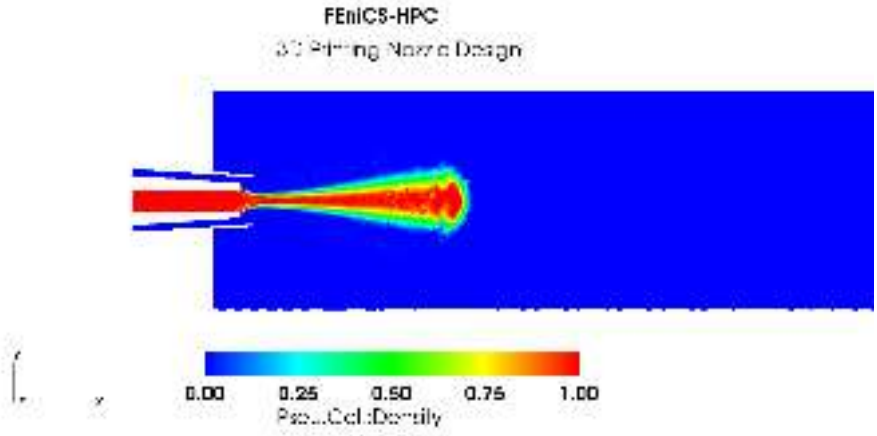


Figure 3.31: Pseu.Col.:Density; viscosity ν = 1e-04, inner inflow u_{inner} = 0.75 and sheath inflow u_{sheath} = 3.75.

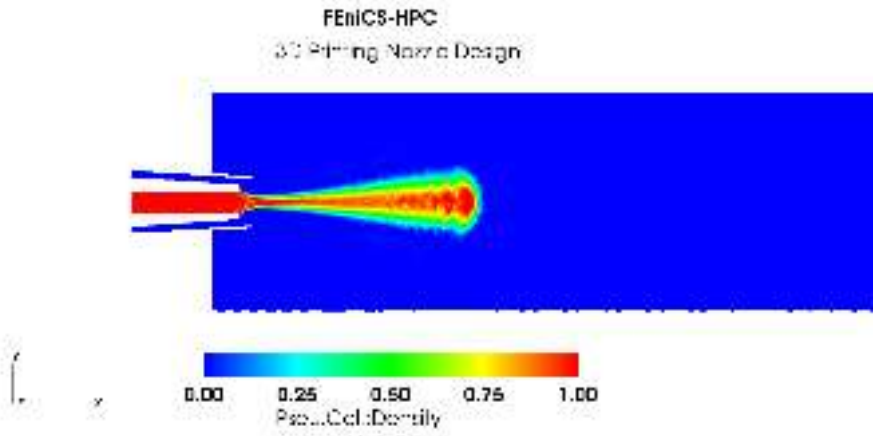


Figure 3.32: Pseu.Col.:Density; viscosity ν = 1e-04, inner inflow u_{inner} = 0.75 and sheath inflow u_{sheath} = 4.75.

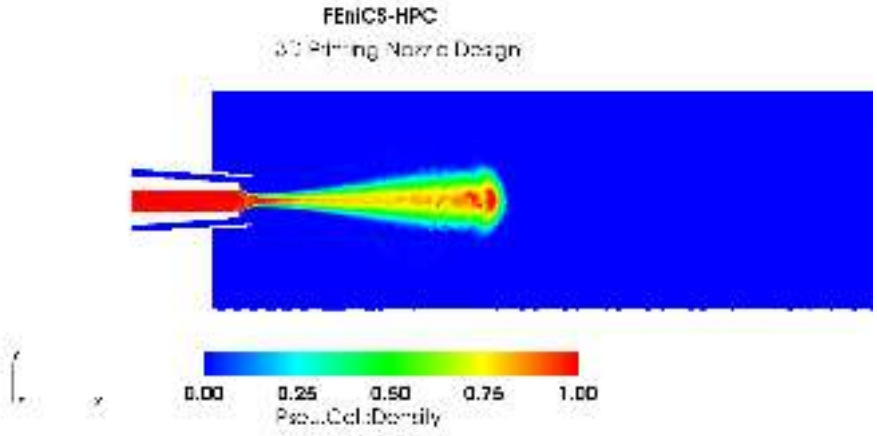


Figure 3.33: Pseu.Col.:Density; viscosity ν = 1e-04, inner inflow u_{inner} = 0.75 and sheath inflow u_{sheath} = 5.75.

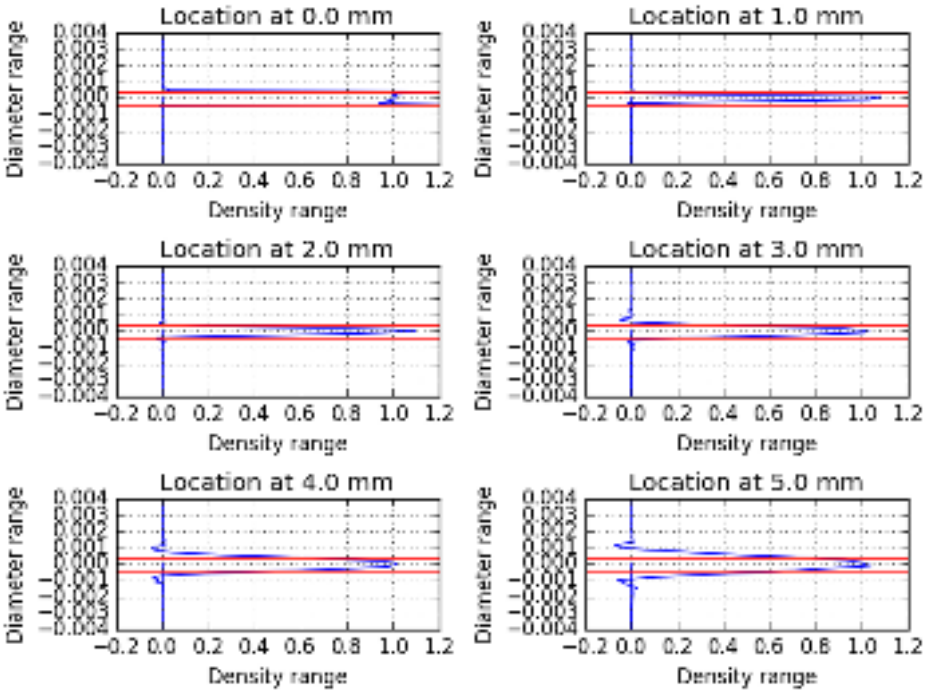


Figure 3.34: viscosity ν = 1e-04, inner inflow $u_{inner} = 0.75$ and sheath inflow $u_{sheath} = 3.75$.

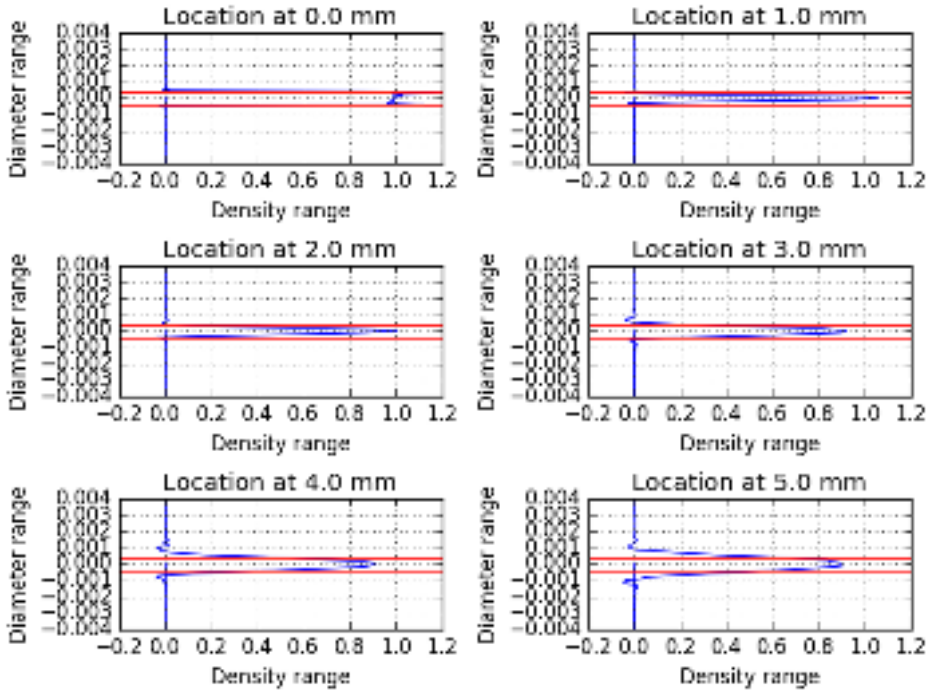


Figure 3.35: viscosity ν = 1e-04, inner inflow $u_{inner} = 0.75$ and sheath inflow $u_{sheath} = 4.75$.

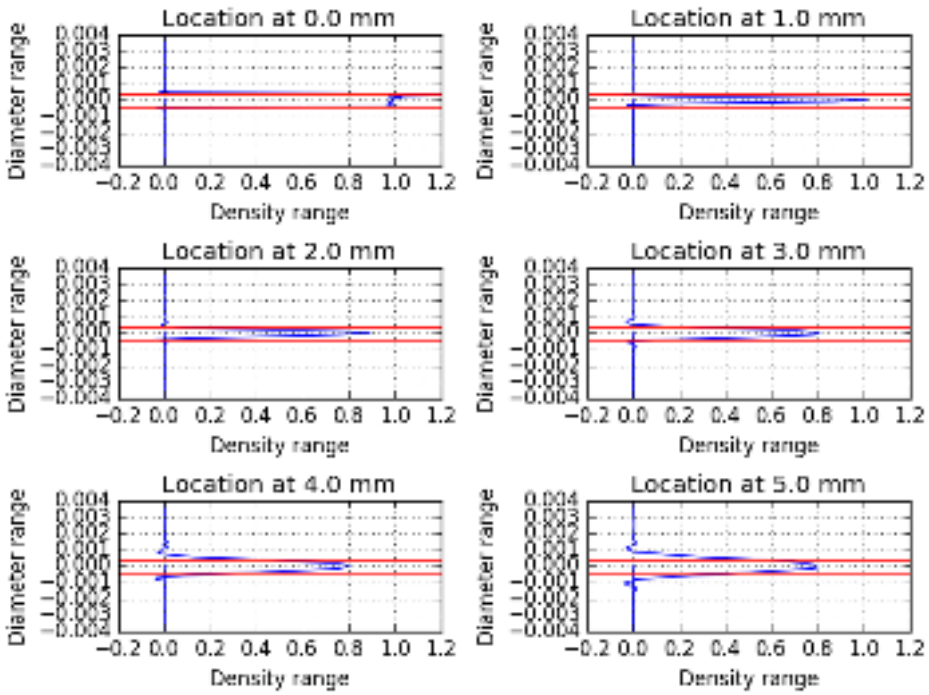


Figure 3.36: viscosity ν = 1e-04, inner inflow $u_{inner} = 0.75$ and sheath inflow $u_{sheath} = 5.75$.

3.3.7 Conclusions

From our simulation results, we see that the dominant parameter, aside from the viscosity, is the sheath velocity. The geometry of the nozzle appears to have less importance. We have thus focused on studying the sheath inflow speed in this report.

In the figures 3.34, 3.35, and 3.36 we see that as the speed of the sheath flow is increased, the width of the inert gas and particle mixture jet is decreased. The parameters corresponding to 3.36 appear to give the best results among the studied cases.

Chapter 4

Parallel visualization, cloud computing and pre-processing

Chapter 4 explains the parallel visualization for the DFS simulation, running the DFS simulation on the cloud computing platform and recommends a suitable tool for pre-processing.

4.1 Visualization

Post-processing is important to visualize the output data from the scientific simulations, and it converts scientific data into *visual form*. Visualization can be described as a pipeline with a dataflow network in which computations are executed as modules that are connected in a directed graph, which describes how the data moves between the modules. There are three types of modules; they are sources, filters, and sinks [71]. Visualization helps to understand the 1D/2D data and multidimensional data set (for example, 3D data set).

4.1.1 Introduction

We will discuss here the Open Source visualization tool VisIt [3] and Paraview [72], which is an interactive, scalable, visualization, animation, and analysis tool. There are also other proprietary tools are available, but they could be expensive or could have limitations with machines or computational cores, for example, tecplot [73]. Both VisIt and Paraview are based on the Visualization ToolKit (VTK) [2] is an open-source software for 3D graphics, image processing, and visualization. VTK does not have a GUI interface, but VisIt and Paraview have a GUI to act as end user friendly for the Visualization. Figure 4.1 shows the typical workflow of the data processing in VTK; this is what is happening when someone uses the VisIt or Paraview.

The functionality of the VTK is as follows:

- VTK has many filters that manipulate the data.
- Can support any kind of data structure.
- Visualization for the 2D plots and charts.
- Good scalability with support of parallel processing (both CPU and GPU).
- Full framework is written in C++ with libraries.

VisIt and Paraview are having the following characteristics:

- Handling the scalar and vector data.
- Support for the structured and unstructured mesh for the 2D and 3D data.
- Allow visualizing the data up to Terabytes (TBs).
- Massive scalability with supercomputers.
- Interactive manipulation(GUI).
- Parallel I/O.
- Availability of the scripting language/(Python and Java).
- Support generic data format (NetCDF, GADGET, CGNS, msh, Silo, BOV, HDF5/XDMF. VTK, etc.).
- Lots of visualization features (isosurface, isovolume, clip, slice, etc.).
- Extract quantitative information.
- Support for the expressions (math, logical, relational, etc.)
- Create movies and animation with high resolution.

Apart from VisIt and Paraview, there are also other open source tools available such as follows (only a few of them listed below):

- Mayavi [74]:
  - able to visualize the scalar, vector, and tensor data in 2D and 3D format.
  - support different data format.
  - python scripting.
- VMD [75]: analyzing large biomolecular simulation data in 2D and 3D format.
- Gnuplot: can plot the 2D and 3D data set.

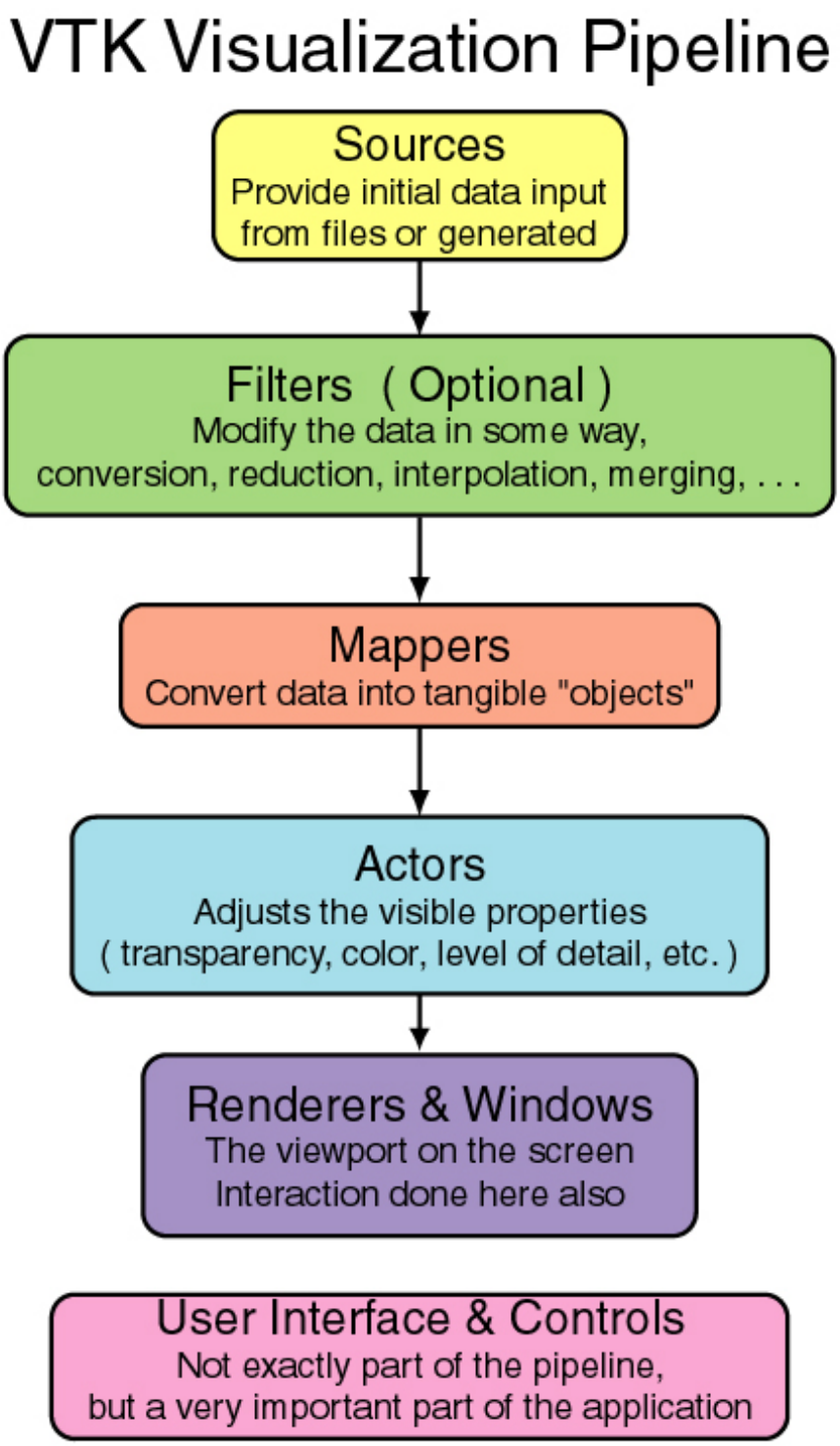


Figure 4.1: Workflow of the VTK pipeline [2].

4.1.2 Work flow of VisIt and Paraview

A typical workflow of VisIt and Paraview consists as follows:

1. Manual visualization of the local workstation.
2. Remote visualization through Graphical User Interface (GUI) from host to a remote computer.

4.1.2.1 Visualization in using local workstation

Once the simulation is done in a supercomputer or in a cluster, output of a scientific data can be downloaded to a local computer for a scientific data visualization. This data transfer can be done through either using the GUI interface between local a computer or with Secure Copy Protocol (SCP).

- Using the GUI and analyze the data and produce images and videos. Figure 4.2 and 4.4 show the GUI interface for both VisIt and Paraview, how data can be analyzed interactively.
- Figure 4.3 shows the recording the python script of the action that we do while using the GUI for the VisIt. Same goes for the paraview as well, Figure 4.5 and 4.6 show events recording and finishing. Recorded events will be documented and executed next time from the command line without opening the GUI.
- *visit -cli -s scriptname.py* for VisIt from command line.
- *pvparaview scriptname.py* for the Paraview from command line.

The above mentioned method is quite useful for the small data set and also for recording some events. But if the data is going to be large, we might need to have lots of time to visualize the data. Or sometimes, our local computer will not even have enough memory, and transferring the data from remote to local will take lots of time. Above all, we can not do parallel visualization. The next Subsection 4.1.3 explains the solution to this problem.

4.1.2.2 Client-server mode

In this method, the simulated output scientific data does not need to be transferred to the local computer (the end user who wants to visualize the scientific data). Figure 4.7, shows the typical work flow of client server mode for the VisIt. Here the local computer is connected to the remote machine (in this case supercomputer) where the computed data is stored. Once the visualization result is obtained, that will be transferred into the local machine.

We can also choose here the parallel visualization, which is also can be seen in Figure 4.7; this means that we can process the data parallelly by using the

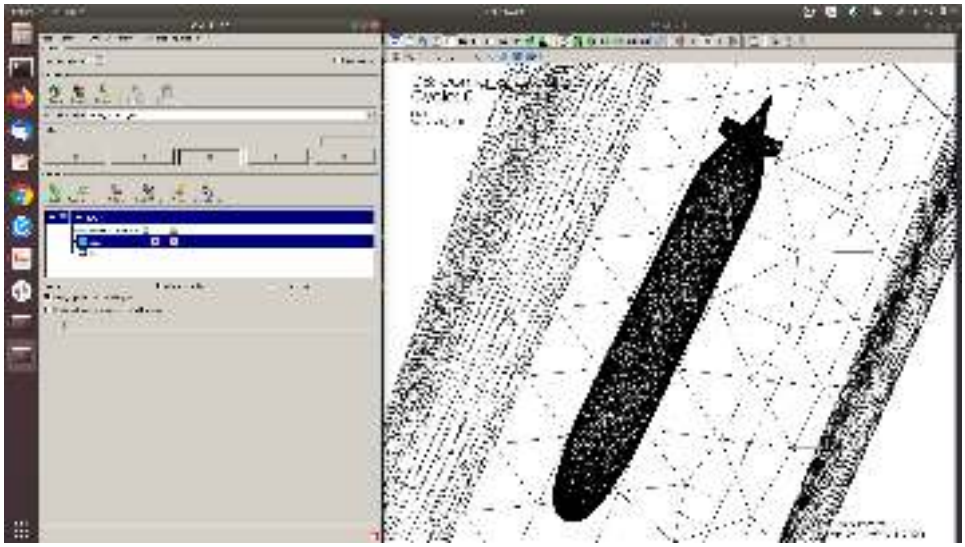


Figure 4.2: VisIt GUI.

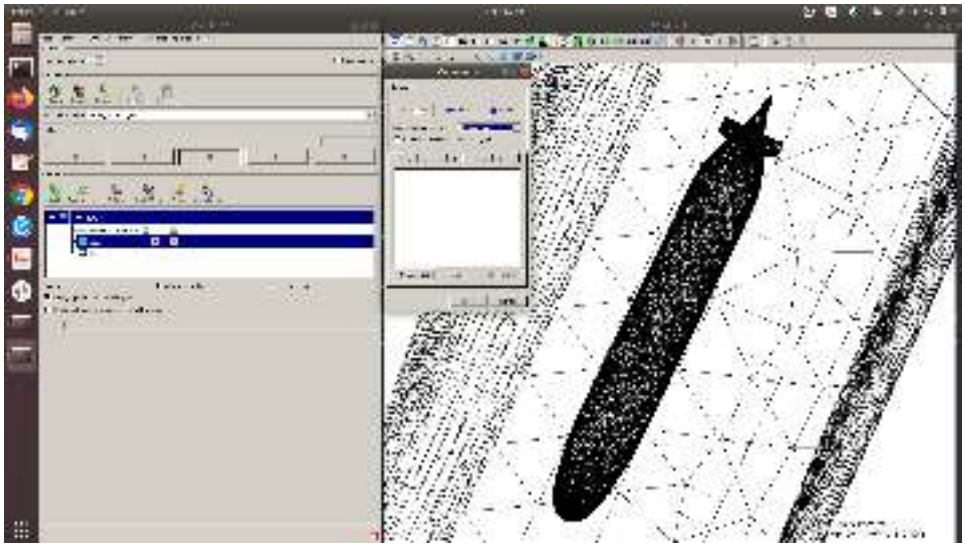


Figure 4.3: VisIt events recording using the python script.

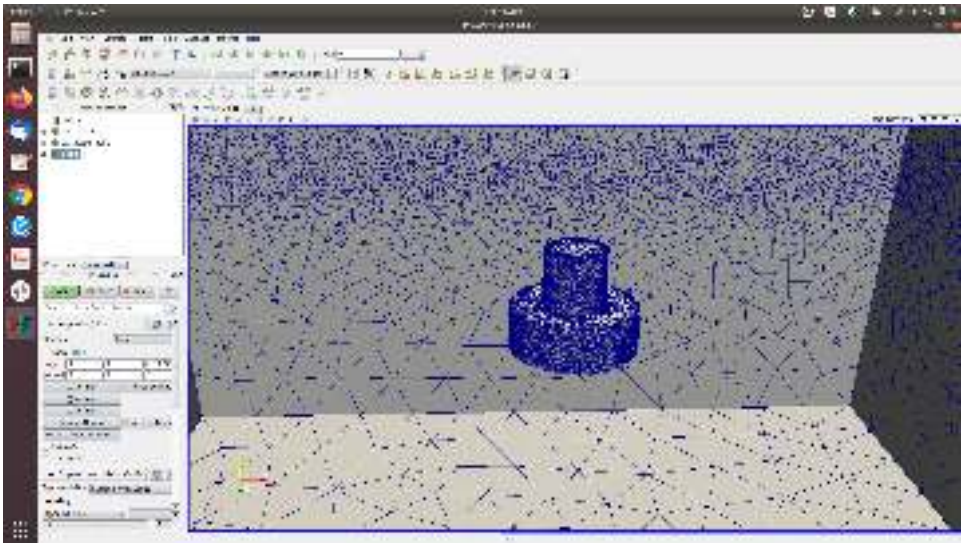


Figure 4.4: Paraview GUI.

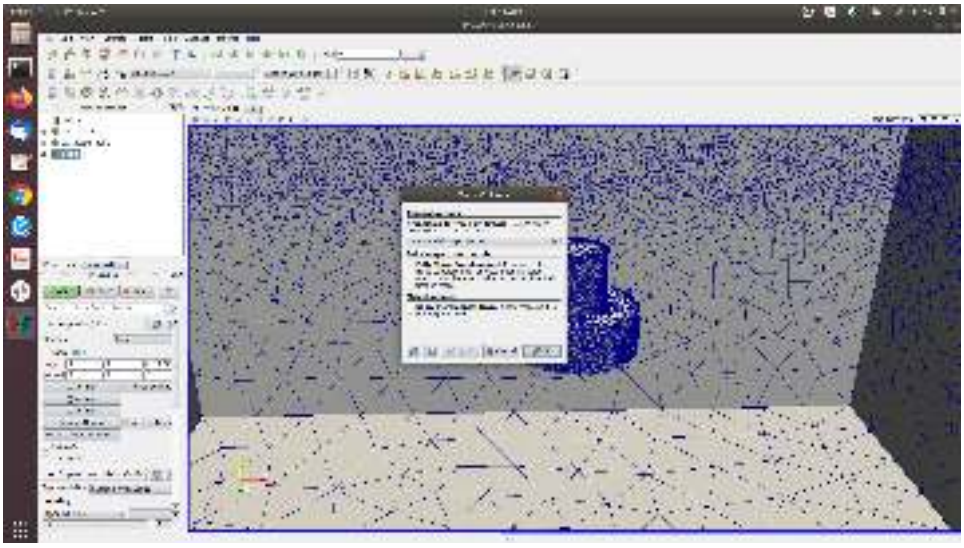


Figure 4.5: Paraview events recording starting.

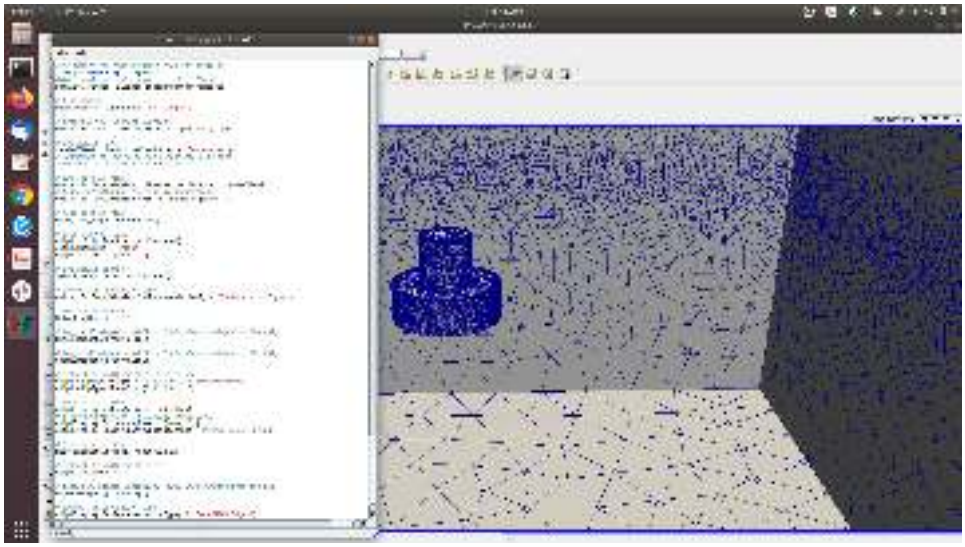


Figure 4.6: Paraview events recording finishing.

supercomputer. Once the results are obtained, it will appear on the local machine. Depends on the data size, one can choose the MPI cores. Both Visit and Paraview support this client-server model approach.

This seems to be handing larger data and parallel data processing. But still, it depends on network connectivity and cumbersome actions. The next Subsection 4.1.3 explains the solution for this problem.

4.1.3 Visualization based on the task parallelization

There is an alternative approach to the above mentioned two methods. That is called the *visualization based on the task parallelization*. We specifically say here the task parallelization because of the FEniCS-HPC simulation output files. Since our methodology based on the adaptivity (and mesh refinement is done only where it is necessary based on the force quantity marker), thus we do not need to do the mesh refinement manually. Basically, the mesh refinement is done where it is needed; in this case, each and every time step file will not be large. Basically, each and every MPI cores can process the single time step files.

Whereas, if your file is very large, you can still do that data parallelization. That means, VisIt or Paraview decompose the data automatically and process it across the MI cores, depends on how much the MPI core allocated during the batch job launcher. Figure 4.10 shows the data parallelization (approach 1) and task parallelization (for FEniCS-HPC) (approach 2).

The following steps describe the process:

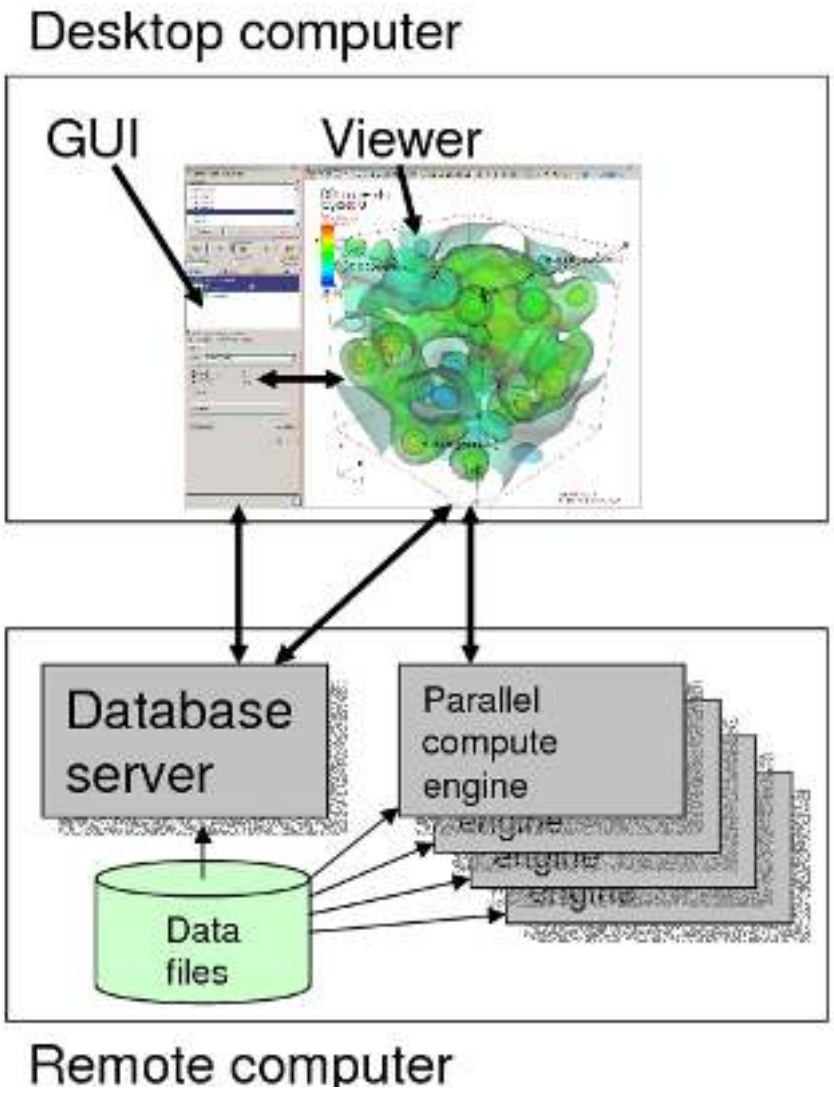
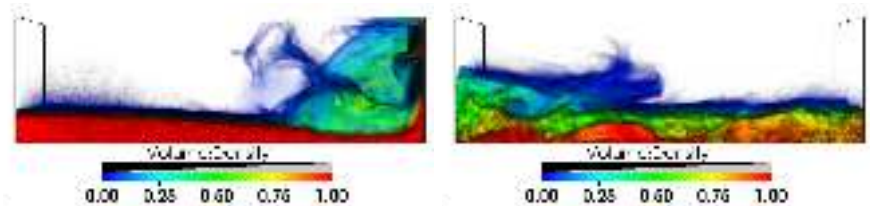


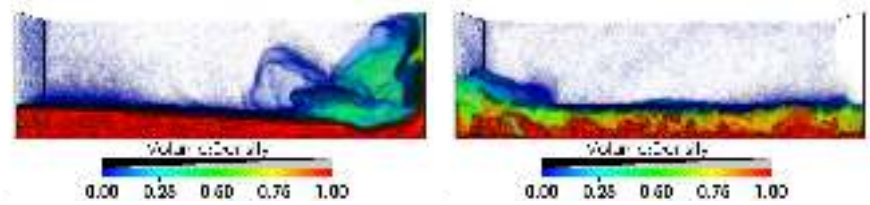
Figure 4.7: Client-server mode [3].

- Record the events for the test case at the local computer.
- Launch the batch scripts across the MPI (or accelerators such as GPUs) cores in the super computer.



(a) sim. on 120 sec. **(b)** sim. on 350 sec.

Figure 4.8: Marin simulation with no phase separation.



(a) sim. on 120 sec. **(b)** sim. on 350 sec.

Figure 4.9: Marin simulation with phase separation.

As an example, one time sample file of the Marin simulation is ≈ 170M and the PDC/KTH Tegner single node memory size is 512 GB RAM. This shows the feasibility of task parallelism with one sample per core. Figure 4.10 shows the difference between the task and data parallelism. Approach 1 shows the data parallelism with MPI, and approach 2 is distributing each time sample to one core. Approach 2 is much faster than approach 1. Figures 4.8 and 4.9 show the volume rendering of the multiphase flow with approach 2. For task parallelism, 54 cores take ≈ 2 hours for 800 time sample files, whereas for data parallelism, 54 cores take ≈ 20 hours to process the 800 time sample files.

4.1.4 Conclusion and future work

In this visualization section, we have described the gentle introduction of the visualization, tools, and methodology. And as of our methodology so called *task parallelization* seems to be an excellent solution to process the data parallelly and efficiently. Another approach can be useful, which is called the In-situ visualization [76], where each and every time step files can be easily visualized before next

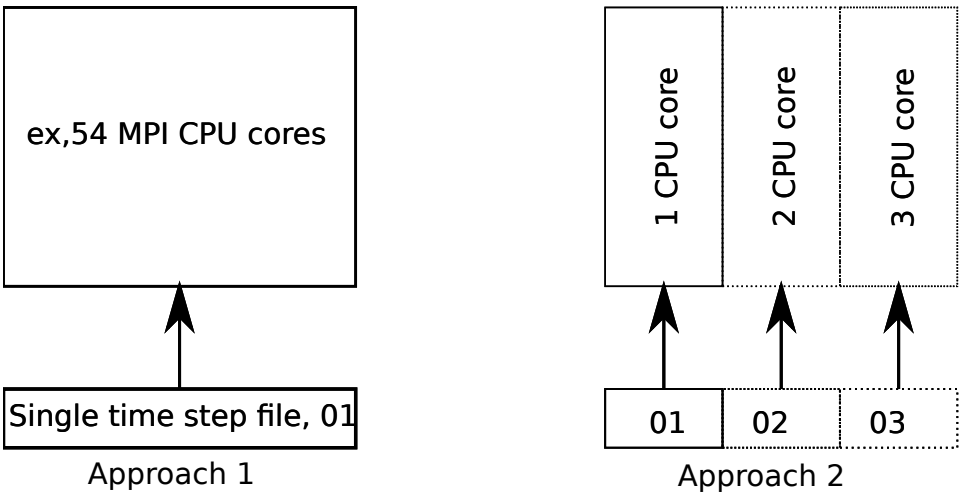


Figure 4.10: Task paralleization for the FEniCS-HPC simulation.

time step is being computed in the simulation. But it might also slow down the pure computational part for simulation. So, in the end, it depends on what we want and what is our interest.

4.2 Cloud computing

Cloud computing means any one can do their computation on remote machine via internet; typically these machines are located in many places and connected to each other. Basically, this service is maintained by the IT expertise of that organization. There are many companies provide cloud service, for example, Amazon Web Services (AWS) [77], Google Cloud [78], Microsoft Azure [79] and IBM Cloud [80].

4.2.1 Cloud architecture

A Cloud computing architecture is organized as follows, (i) Software as a Service (SaaS) (ii) Platform as a Service (PaaS) (iii) Infrastructure as a Service (IaaS), see in Figure 4.11. Among these, IaaS is quite useful for the small firm or companies who does not want to invest large money for the hardware that they needed. The others such as SaaS and PaaS useful for companies such as Yahoo, Hotmail, Gmail, and software maintenance or software provider companies.

4.2.2 Why Cloud Computing

The following points describe the importance of the cloud computing.

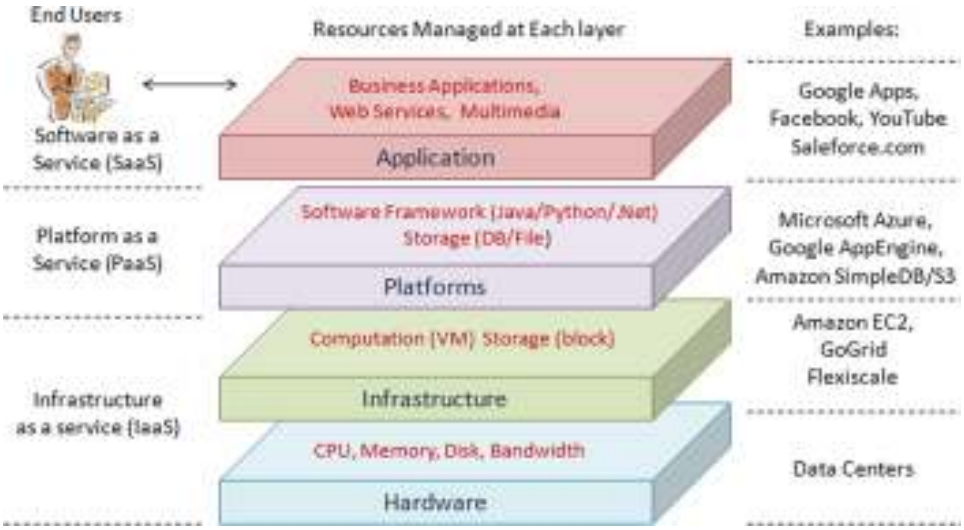


Figure 4.11: Work flow diagram of cloud computing architecture [4].

- Since the hardware is maintained by the firm or company who owns it, as an end user we do not need to pay any money for the maintenance. Of course, this cost will be indirectly included in the IaaS. But many not be much, since we use the resources from time to time.

- Amazon HPC promises scalable architecture. This means that, more scalable computational cores with a good network connection.

- There is no queue time compared to a traditional supercomputer. Basically on the traditional supercomputer, one should remain in the queue for some time before the computation starts.

- And we get to access the latest GPU accelerators in the cloud such as Nvidia Pascal and Volta GPUs [81]. Not only just latest GPUs plus also latest powerful CPUs as well.

- For example, High-performance computing (HPC), low communication latency required (FEM, applications in CFD and weather forecasting) [82].

- Another example, High-throughput computing (HTC), high communication latency acceptable (for example, media rendering, transcoding and genomics) [82].

- Amazon network appears competitive to Cray architecture: in-house ethernet-type, 100Gb. Avoid noisy neighbor and dedicated network links [83].

4.2.3 Google Compute Engine

Here, we will focus on launching the Google Compute Engine from the users local computer.

- There are a number of commercial and open-source tools that can be used to launch or create compute nodes in the Google Cloud from the users local computer.
- We would like to list out two well known open-source software can be used to create a compute node on the Google Cloud. One is called Kubernetes [84] which is within the Google framework. And another one is called Elasticluster [85].
- Here, we will focus on the Elasticluster, because it can be used for both Amazon and Google Cloud.
- FEniCS-HPC is a good candidate for the cloud solution, since our simulation relies on the adaptivity, the computation can be done using limited computational power. And also the accuracy and goal oriented functions are set by the user, that makes computation even faster or optimized.

As we have mentioned above that we will focus on the Elasticluster, the following steps describe the functionality of the Elasticluster.

- Using the Elasticluster we can easily create a cluster/supercomputer environment in the Google Compute Engine (GCE).
- Instance creation, launching the computation, monitoring job can be done via terminal from users local computer.
- Only few steps needed to install Elasticluster on end users local computer.
- Create a familiar working environment for the end users as they work in the supercomputer.
- Both computation and post-processing can be done in GCE via Elasticluster.

4.2.4 Compute Cluster Creation in the GCE

The following steps provide how one can create compute cluster in the GCE using the Elasticluster.

- Install the elasticluster on the local computer [86].
- Allow the Elsticluster to access your CCP project [87].
- Generate an SSH key pair [87].

```
Your cluster is ready!

Cluster name:     myslurmcluster
Cluster template: myslurmcluster
Default ssh to node: frontend001
- compute nodes: 5
- frontend nodes: 1

To sign in to the frontend node, run the following command:

elasticluster ssh myslurmcluster

To upload or download files to the cluster, use the command:

elasticluster sftp myslurmcluster
```

Figure 4.12: Elasticluster created the cluter/supercomputer environment.

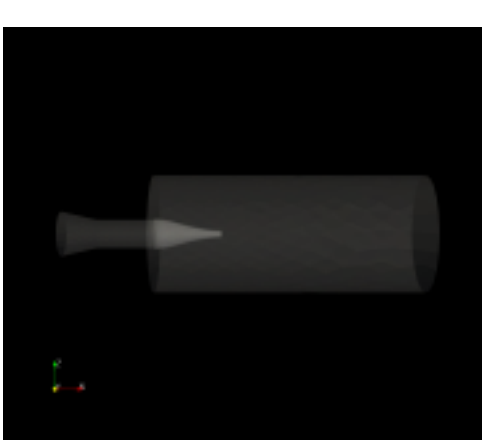
- Configure Elasticluster (specify the login and compute node processors, memory, and operating system.
- Elasticluster start myslurmcluster (this will create the clustering environment in the cloud [88].
- Figure 4.12 shows the 5 compute node with 1 front end node of cluster in GCE from the local computer using the Elasticluster.

4.2.5 FEniCS-HPC on the Google Cloud

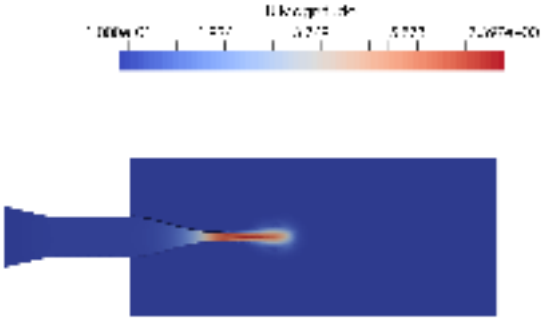
We have tested a simple test case for the flow through the nozzle. Which is a small project connected to the GENTALVE project from Bilbao, Spain. The idea here is that the probe will be placed at the tip of the nozzle. But before that, we would like to see how the flow field in the fluid domain would look like. Figure 4.13 shows the probe geometry and simulation at various time steps. And also, table 4.1 shows the simulation time comparison between the Beskow and Google cloud.

4.2.6 Conclusion and future work

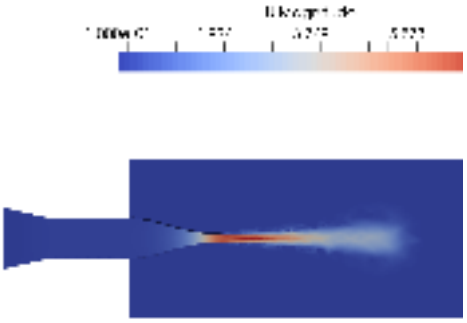
As we can see in here (see table 4.1) that Google cloud is a bit slower than the Beskow. Both simulations tested on the single node with MPI processor at both



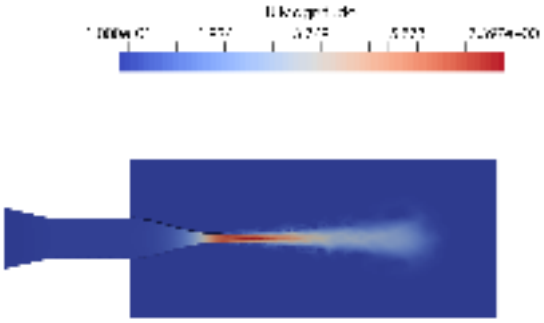
(a) Probe Geometry.



(b) Simulation at 0.04 sec.



(c) Simulation at 1.6 sec.



(d) Simulation at 2 sec.

Figure 4.13: Probe model geometry and simulation at various timestep.

Adapt.iterations	Vertices	Elements	CPU core hours (Beskow KTH)	CPU core hours (Google Cloud)
Adapt.iteration 0	16612	88862	4	4.15
Adapt.iteration 1	24123	12750	6	6.10
Adapt.iteration 2	31641	169154	8	8.20
Adapt.iteration 3	41210	220966	10	10.05
Adapt.iteration 4	52881	285231	12	12.35

Table 4.1: Performance comparison between Beskow and Google cloud.

platform. Whereas, we know that for sure, the single node at Beskow processors within the node, so the communication here is very fast on the other hand, it is hard to see how processors are connected in the google cloud. But still, it is very hard to come to a conclusion at the moment. For the clean comparison, more tested is needed with bigger simulation and using more MPI processors.

4.3 Pre-processing

In the CFD community, pre-processing is a time consuming work, and one should needs to have special knowledge and lots of practice to create structured or unstructured mesh for the simulation, especially for the RANS/LES simulation. The standard procedure in the CFD community is that manually refine the mesh around the surface of the domain and downstream side of the flow to capture the drag, turbulence, and wake. Most of the cases, we do not know where it will appear in the simulation, so to avoid the numerical error, people usually do most of the time, complete mesh refinement in the downstream up to some extent in the computational domain. And boundary layer refinement around the object. Figure 4.14 shows the typical example of manual mesh refinement in the simulation domain. As we can see here, the boudary layer around the Ahmed body is finely refined, and also downstream of the flow is refined.

4.3.1 Tools for the pre-processing

There are a number of commercial and open-source meshing tools are available. Most of the cases, the open-source tools are useful for simple geometry. The Salome [89] and Gmsh [90] are widely used open-source meshing tool in the CFD community; and Pointwise [91] and ANSA [92] are popular commercial pre-processing tools. When it comes to complex geometry, it is better to go with commercial tools. In this thesis, we have used the commercial meshing tool called ANSYS [6].

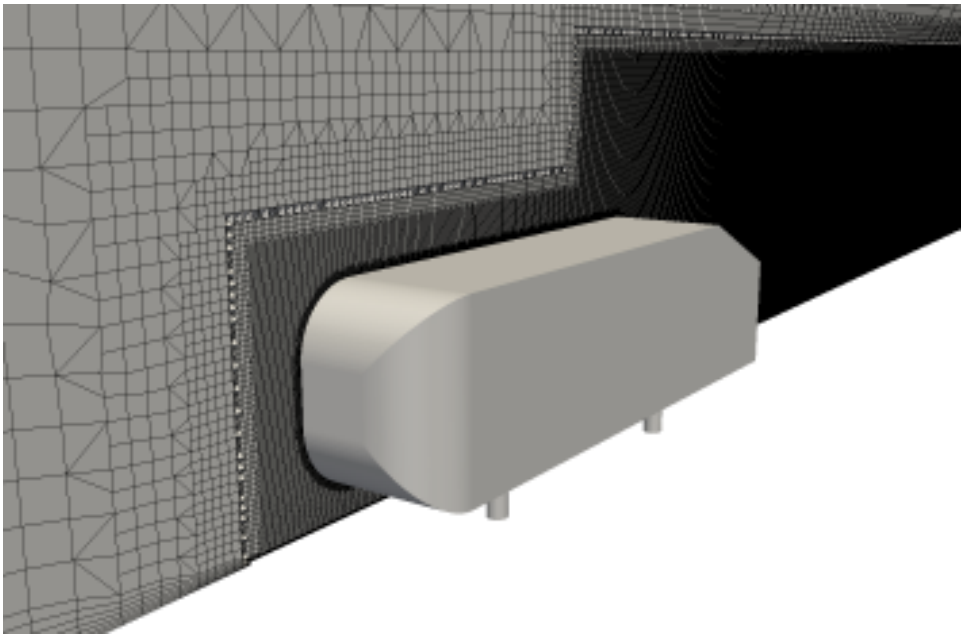


Figure 4.14: Manually refined the mesh flow around the Ahmed body [5].

4.3.2 Recommendation: pre-processing for DFS

ANSYS has a quick and smart algorithm for the tetrahedron elements. It has a two algorithm are as follows:

- Patch confronting.
  - Meshing is starting from the edges, faces, and finally to the volume.
  - If the CAD is clean, the meshing will be good.
  - Meshing size can be defined globally and locally.
- Path independent.
  - First, volume mesh is created, then to surface and edges. Completely opposite to patch confronting method.
  - Good for bad quality CAD.
  - Method has sizing details.

In this thesis, all the CAD model and meshing is done by using the ANSYS CAD/Meshing with patch confronting. Which is quite easy to use with minimal effort to get good mesh for DFS. Where we need to refine the mesh to capture the CAD detail.

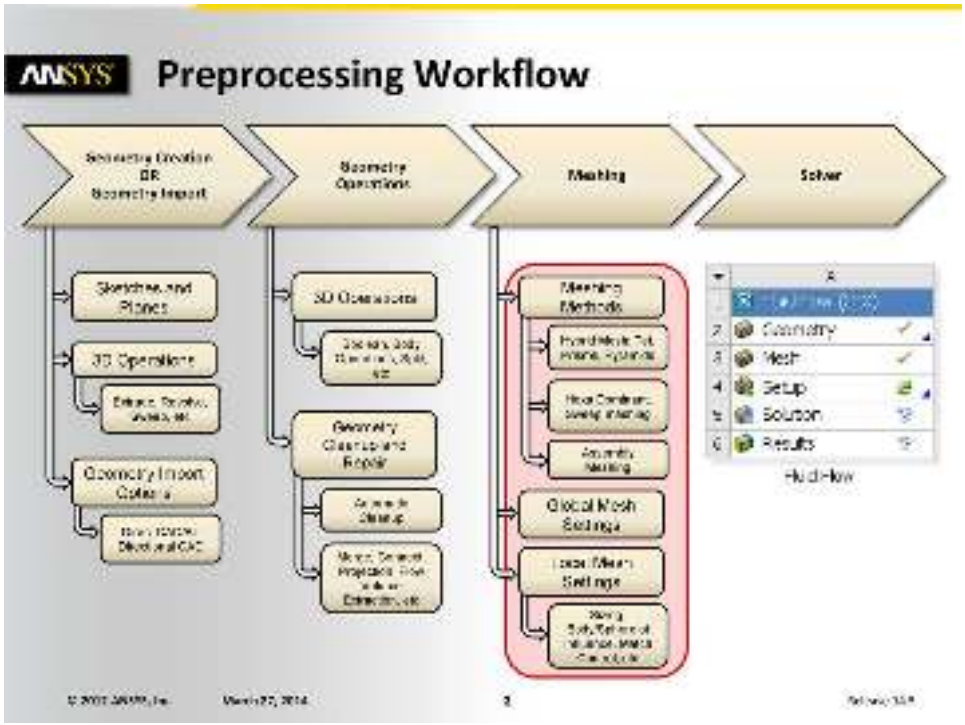


Figure 4.15: Workflow of ANSYS [6].

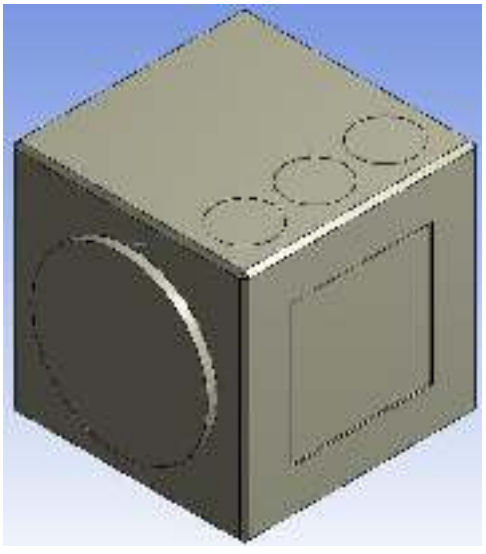
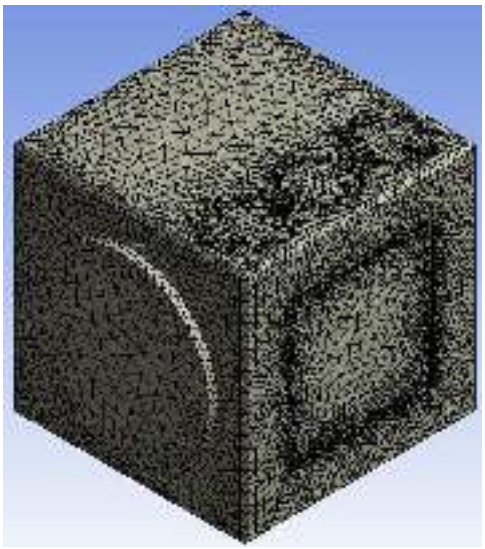
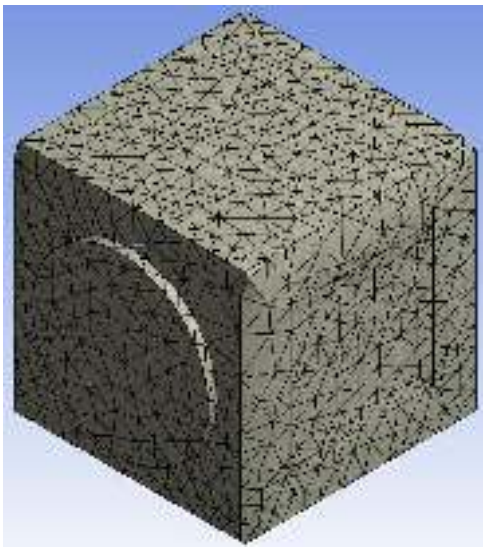


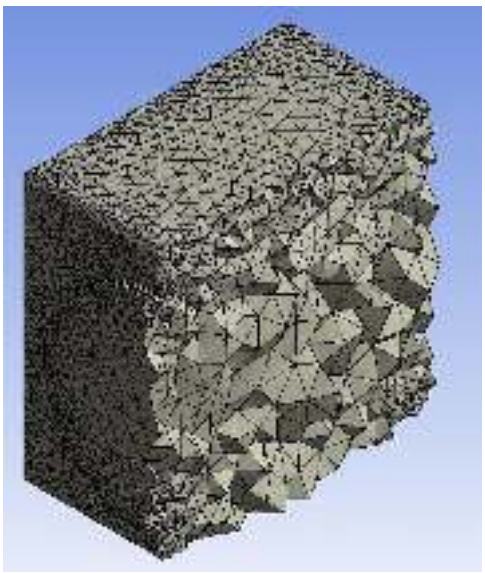
Figure 4.16: CAD model contains small detial [6].



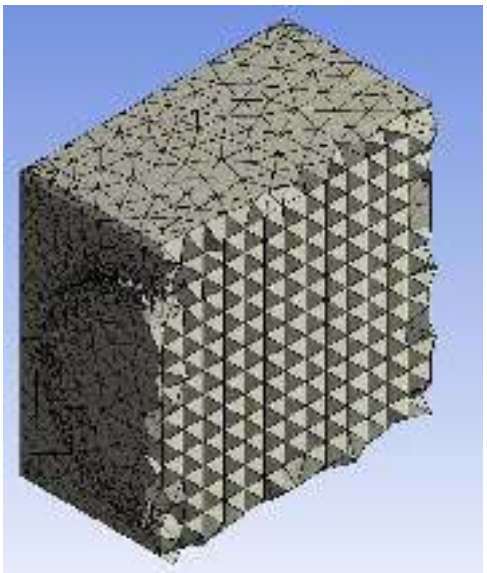
(a) Patch confronting: all geometric details are captured [6].



(b) Patch independent: ignores the geometry detial [6].



(c) Patch confronting slice: delaunay mesh-smooth growth rate [6].



(d) Patch independent slice: octree mesh with approximate growth rate [6].

Figure 4.17: ANSYS meshing methodology for patch confronting and patch independent.

Chapter 5

High performance computing

This Chapter 5 starts with explaining the architecture of CPU and GPU. Later, it shows the numerical stencil implementation on multiple GPU and Jacobi implementation on the Kalray architecture aiming at exascale computing.

5.1 Introduction

The concept of parallel computing was discussed even in the 19th century when there were no electronic machines available, as Babbage*et al.* [93] tried to perform multiplication of two numbers using their Difference Machine.

The beginning of the high performance computing era, the Von Neumann architecture, attempts to illustrate the concept of sequential computing shown in Figure 5.1. Although the computation can be done fast the I/O has a problem with the memory, which is called Von Neumann bottleneck [94]. In recent years, this problem has been reduced drastically using memory banks that provide parallel I/O memory. Today, efficient parallelization is achieved by vectorization, multiple processors, and accelerators (for example, GPU and CPU co-processors). The efficiency of a computer can be measured by FLoating point Operations Per Second (FLOPS). Supercomputers are ranked based on how fast they can solve the LINPACK benchmark, which consists of dense linear algebra.

5.2 CPU architecture

Today, all CPUs have multiple cores. For example, Intel Xeon Gold has 16 CPU cores and Figure 5.2 shows the standard multi-core CPU concept. This trend might increase gradually, which means the more production of CPU cores would take place over the years as technology evolves. In Figure 5.2, L1 cache is private to the cores, and L2 cache is shared among the cores, whereas the last level memory is shared between all the cores by memory interface. Evidently, to achieve maximum performance of multicore, one needs to do multi-threading or multi-processing.

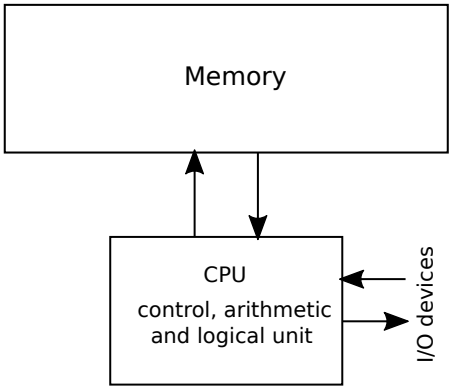


Figure 5.1: The Von Neumann architecture.

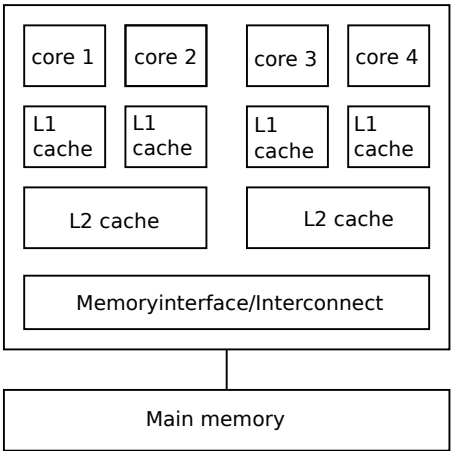


Figure 5.2: Standard multi core CPU.

5.3 Parallel architecture

Parallel architectures are classified into three components, which are, control structure, memory organization, and network connection topology.

5.3.1 Control structure

Michael J. Flynn divided computer architectures into four categories [95, 96]. They are as follows:

- Single instruction stream single data stream (SISD): which is based on the Von Neumann architecture, i.e., simple, sequential, and uniprocessor.
- Single instruction stream, multiple data streams (SIMD): only one single stream of instruction can be executed, and, moreover, as a single instruction can be used to process multiple data, this leads to *data parallelization*. Figure 5.3 shows the *SIMD* concept of simultaneous instructions and single instruction with multiple data.
- Multiple instruction streams, single data stream (MISD): in reality, it is not useful since there is no program that can readily map into MISD organization.
- Multiple instruction streams, multiple data streams (MIMD): multiple instructions can be executed on multiple data streams. For example, on multicore CPUs, a processor can execute the same code or different branches of the same code. This is typically done by *thread level parallelization* on multicore CPUs. It can be hard to debug the code.

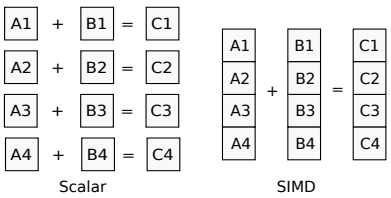


Figure 5.3: SIMD model.

5.3.2 Memory organization

Memory hierarchy is a key concept in standard and parallel architectures. Figure 5.4 shows a basic memory hierarchy of computer. Cache is the fastest access memory compared to off-chip memory. Cache holds the temporal information from the main or secondary memory, which might be currently used by the processor [97].

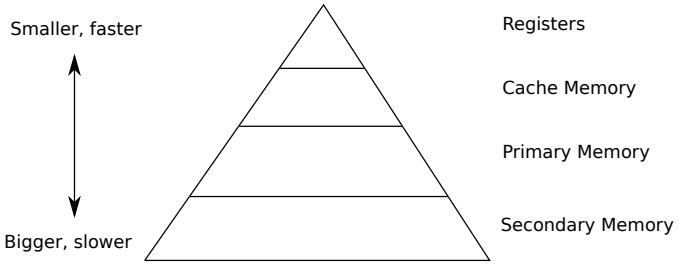


Figure 5.4: Memory hierarchy.

- Shared memory: all the processors have the same address space, which can share and access the data from the main memory. Shared memory architecture is classified into three variants, based on how the bus network connects the memory and processor. Figure 5.4 shows 3 different types of shared memory architecture.
- Distributed memory: each processor has it's own local memory, which means no global memory shared by the processors. The local memory is accessed by the physical network to transfer data to and from another processor.
- Hybrid memory system: A distributed memory with a shared memory approach in each cluster. This typically consists of multicore CPUs with/without an accelerator such as GPUs.

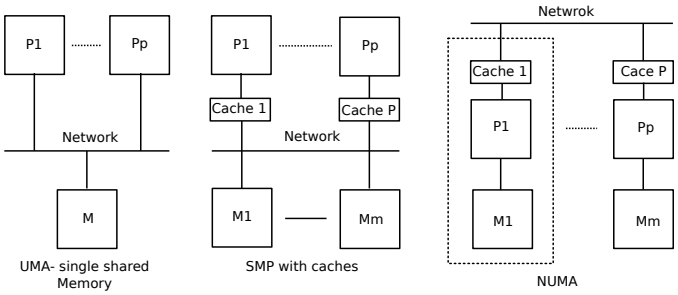


Figure 5.5: Shared memory architectures.

5.3.3 Network topology

Many variants can be used to organize the processors in an the efficient way how the processors are connected to each other. For example, bus, linear array, ring, 2D grid, torus, 3D grid, 3D torus, tree, and aeries. Each of these network topologies has uts advantages and disadvantages; for example, tree network has a higher cost but reduces the bandwidth bottleneck.

5.4 GPU architecture

A key difference between GPU and CPU is that a GPU has a large number of cores, for example, the latest Nvidia Volta GPU has 2560 (FP64 Cores) cores [98]. On the other hand, a CPU has a higher clock speed than a GPU.

GPUs are one of the main accelerators in the HPC field, and almost all supercomputers in the world have a GPU accelerator. In order to achieve exascale computations, GPUs appear to be a good option. Today, there are also PDE frameworks with unstructured meshes that demonstrate good performance with GPUs, such as high-order PyFR.

Processing elements in the GPU are grouped together and called Streaming Multiprocessor (SM). Multiple threads can be executed concurrently on each and every SMs, and this is based on Single Instruction Multiple Thread (SIMT) architecture. These threads are grouped (e.g. into groups of 32), and executed across the SMs. These grouped threads are called "warps" and they have their own registers and instructions. GPUs allow scalable multi thread arrays, which means grid and thread blocks can be in 1D, 2D, and 3D. On CUDA, this thread block is called a cooperative thread array (CTA) [99]. The Performance of the GPU can be improved if the "latency" is reduced [100]. Here, latency refers to the number of clock cycles needed to execute the warps in the SMs [101].

5.4.1 Memory organization in GPU

A GPU has a similar memory organization as a CPU, but with some key differences as follows.

`Register:` is the fastest memory; each thread owns registers.

`Local:` holds the register spelling and is cached in typically the next level of memory hierarchy such as L1 and L2 cache.

`Shared Memory:` for both Fermi and Kepler, shared memory is configurable by the programmer, and the usual size is 64 KB. However, with the latest Pascal and Volta architecture, L1 and shared memory are merged and each of them then share 64KB each. Each and every SM has its own shared memory.

`L2 cache:` a normal GPU has a single L2 cache is shared by all the SMs in the GPU. The Volta GPU has 6144 KB in size.

Figure 5.6 (left) shows the generic memory access on the CUDA device and (right) shows the thread memory access in the Kepler architecture. Figure 5.1 shows threads, access to memory.

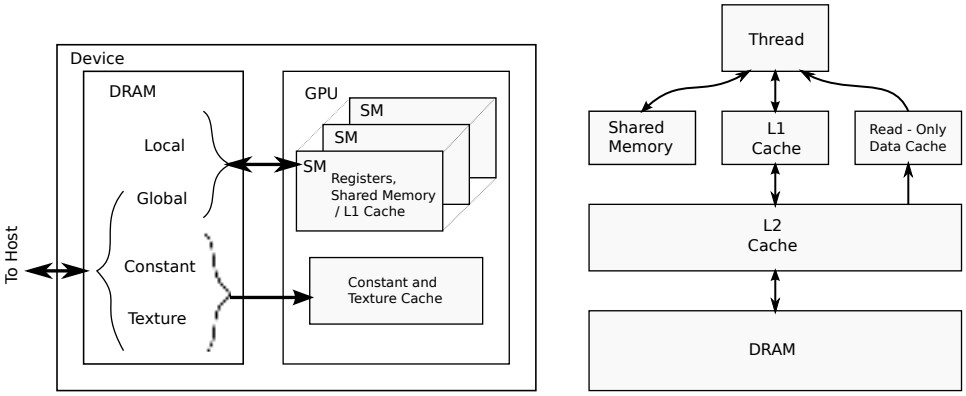


Figure 5.6: left:CUDA device memory spec.; right: Kepler's read cache memory.

Table 5.1: CUDA *threads memory access in the device.*

Memory	Location	Cached	Device Access	Scope	Life Time
Register	On-chip	N/A	R/W	one thread	thread
Local	DRAM	yes**	R/W	one thread	thread
Shared	On-chip	N/A	R/W	all threads in block	thread block
Global	DRAM	*	R/W	all threads in host	Application
Constant	DRAM	Yes	R	all threads in host	Application
Texture	DRAM	Yes	R	all threads in host	Application
** cached* L2 *by default by latest compute capabilities*					
*** cached* L2 *by default only on compute capabilities 5.x*					

5.4.2 Latest advancement in GPUs

CUDA is a programming language used to program Nvidia GPUs. CUDA has compute capabilities, that controls the hardware programming capabilities on the respective Nivida GPU microarchitecture. We would like to point out some advanced technologies that are available at present, as follows:

- NVLINK: the memory transfer between the CPU-GPU was a major bottleneck in the early stage of GPUs, that is before Tesla P100. NVLINK has minimized this bottle gradually, which means that the data transfer between CPU and GPU and among the different GPUs and the bandwidth can achieve up to 300 Gigabytes/second on the second generation of NVLINK with Volta architecture GPU.

- Unified Memory: this is available as a virtual memory whose, CPU and GPU share the same address space. This allows simpler implementation, and it facilitates the fast porting of complex data structure programming models.
- Tensor Cores: Typically AI applications deal with huge data, and most of the times it will be dealing with linear algebra operations. The latest micro architecture GPUs have Tensor cores, with half precision compute capabilities, and the normal cores in the SMs have both single precision and double precision.
- New Concepts: on the Volta and Pascal, SMs are grouped as multiple GPU Processing Clusters (GPC), which means that each and every GPC has one set of SMs and tensor cores.

5.5 Multiple GPU Implementation for Stencil Numerical Computation

To explore the possible performance of multi-GPU computation on a single compute node, we considered a stencil computation, with application in geological folding simulations. Geological folding simulations are carried out to find oil in the ground. A specific reason for focusing on this problem is also so that it requires large computing resources.

Here we show the CUDA programming challenges with using multiple GPUs inside a single machine to carry out plane-by-plane updates in parallel 3D sweeping algorithms. In particular, care must be taken to mask the overhead of various data movements between the GPUs. Multiple OpenMP threads on the CPU side should be combined multiple CUDA streams per GPU to hide the data transfer cost related to the halo computation on each 2D plane. Moreover, the technique of peer-to-peer data motion can be used to reduce the impact of 3D volumetric data shuffles that have to be done between mandatory changes of the grid partitioning. We have investigated the performance improvement of 2- and 4-GPU implementations that are applicable to 3D anisotropic front propagation computations related to geological folding. In comparison with a straightforward multi-GPU implementation, the overall performance improvement due to masking of data movements on four GPUs of the Fermi architecture was 23%. The corresponding improvement obtained on four Kepler GPUs was 47%.

5.5.1 Background

Motivated by higher energy efficiency, a new trend with high-end computing platforms is the adoption of multiple hardware accelerators such as general-purpose GPUs and many-integrated-core coprocessors per compute node. The most prominent example is Tianhe-2, which is the current No. 1 system on the TOP500 list [102]. Three Xeon Phi coprocessors can be found in each of Tianhe-2ś 16,000

compute nodes. With respect to clusters that have multi-GPU nodes, the TSUBAME 2.5 system is known for having three NVIDIA Tesla K20x GPUs per compute node. Together with this new hardware configuration trend, there comes the challenge of programming. In addition to properly offloading portions of a computation to the individual accelerators to achieve higher computing speed, a new issue arises regarding accelerator-to- accelerator data transfers. Although MPI communication should be used between accelerators residing on different compute nodes, intranode data transfers between cohabitant accelerators have the possibility of using low-level APIs that incur less overhead than the MPI counterpart. The hardware context of this paper is using multiple NVIDIA GPUs within the same com- pute node. As the computational domain, we choose to study parallel 3D sweeping algorithms, which have a causality constraint in that 2D planes have to updated one by one in sequence, where parallelism exists between mesh points lying on the same 2D plane. Following our previous work in the simpler scenario of 3D stencil computations [103], we want to investigate how to apply multiple CUDA streams, multiple OpenMP threads and NVIDIA GPUDirect [104] to 3D sweeping algorithms for masking the overhead of GPU-to-GPU data transfers. Another objective of this paper is to quantify the actual performance gain of using multiple GPUs for simulating 3D anisotropic front propagation, compared with using a single GPU [105]. To our knowledge, previous work on using GPUs to simulate anisotropic front propagation all targeted single-GPU platforms, such as [106–108]. Therefore, this paper is also novel with respect to using multiple GPUs for this particular computational problem.

5.5.2 Mathematical background

In this paper, we consider parallel 3D sweeping algorithms that use a Cartesian grid, which is of dimension $(n_x + 2) \times (n_y + 2) \times (n_z + 2)$. The mesh point values, denoted by $T_{i,j,k}$, are iteratively updated by sweeps each being made up of six subsweeps that alternate between the positive and negative x, y and z-directions. Each sub-sweep consists of plane-by-plane updates that move consecutively through the 3D mesh. Computations of the mesh points that lie on a 2D update plane are independent of each other, thus parallelizable. The following pseudocode shows one sub-sweep along the positive x-direction and another sub-sweep in the opposite direction, see Fig 5.7.

for $i = 2 \rightarrow n_x$ **do**
  for all $j \in [1, n_y]$ and $k \in [1, n_z]$ **do**
    Update $T_{i,j,k}$ using values $T_{i-1,j\pm a,k\pm b,} a \in 0, 1, b \in 0, 1$
  end for
end for
for $i = n_x - 1 \rightarrow 1$ **do**
  for all $j \in [1, n_y]$ and $k \in [1, n_z]$ **do**
    Update $T_{i,j,k}$ using values $T_{i+1,j\pm a,k\pm b,} a \in 0, 1, b \in 0, 1$
  end for

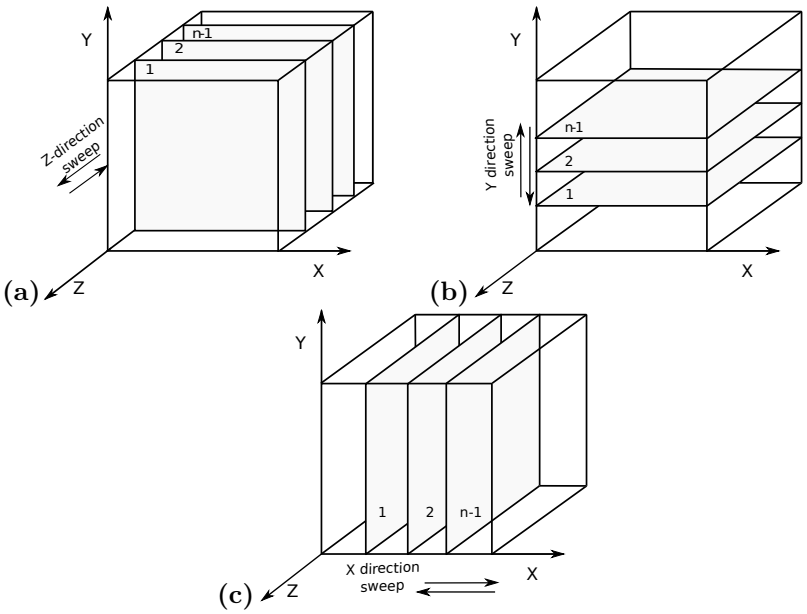


Figure 5.7: 2D plane sub-sweeping (a) in Z-direction, (b) in Y-direction and (c) in X-direction.

end for

The four other sub-sweeps (two in the y-direction and two in the z-directions) are similar. We also remark that the sub-sweeps update only the interior mesh points, i.e., $1 \leq i \leq n_x$, $1 \leq j \leq n_y$, $1 \leq k \leq n_z$. The boundary points assume known solution values. In the above pseudocode, updating a mesh point relies on 9 mesh points that lie on the preceding plane, see figure 5.8 for an example. A concrete application of such a sweeping algorithm can be to simulate anisotropic front propagation that is described by the following static Hamilton-Jacobi equation:

$$\begin{aligned} F(\mathbf{x}) \left\| \nabla T(\mathbf{x}) \right\| + \Psi(\mathbf{a} \cdot \nabla T(\mathbf{x})) = 1, \\ T = t_0 on \Gamma_0, \end{aligned} \tag{5.1}$$

where $T(x)$ can model the first-arrival time of a propagating front that originates from the initial surface Γ_0. When the viscosity solution of 5.1 is used to model geological folding, vector a marks the axial direction of the fold, with F and Ψ being nonzero constants. For details about the mathematical model and the derivation of sweeping-based numerical schemes, we refer the reader to [108] and the references therein.

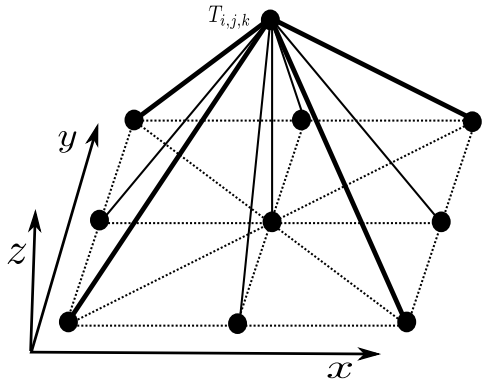


Figure 5.8: An example of data dependency associated with sub-sweeps along the z-direction.

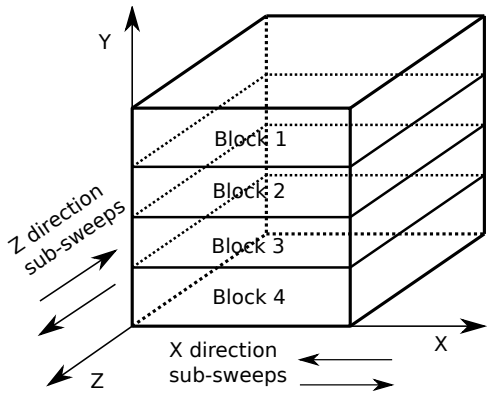


Figure 5.9: A partitioning of the 3D Cartesian grid that suits parallelization of sub-sweeps in both x and z-directions.

5.5.3 Domain decomposition

Parallelism within parallel 3D sweeping algorithms exists among mesh points that lie on the same 2D update plane, but not across the planes. That is, parallelization can be realized by dividing each 2D plane among multiple computing hardware units, such as GPUs. Due to the rectangular shape of a 2D plane, it is natural to assign each GPU with a rectangular subdomain. Moreover, since each sub-sweep moves along a given spatial direction, from the top (or bottom) 2D plane to the bottom (or top) plane, all the 2D planes associated with one sub-sweep can use the same partitioning. This means that, for sub-sweeps along a specific spatial direction, the 3D Cartesian grid should be partitioned by cutting surfaces parallel with the sub-sweeping direction. However, we recall that the sub-sweeps

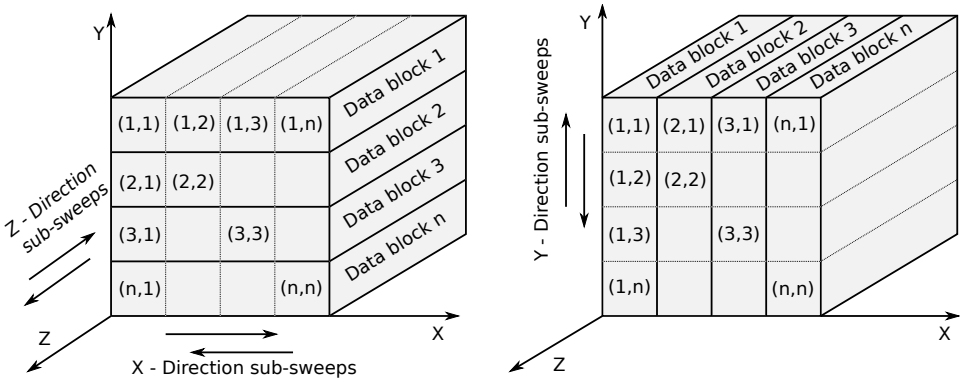


Figure 5.10: An example of volumetric data shuffle in connection with changing the grid partitioning.

alternate between the three spatial directions, thus there does not exist a universal partitioning that works for all x, y and z-directions. The best partitioning strategy is to let sub-sweeps of two spatial directions share one partitioning, switching to another partitioning for the third spatial direction. Figure 5.9 shows a partitioning of the 3D Cartesian grid that can be used to parallelize sub-sweeps in both x and z-directions.

5.5.4 Data transfer

There are two types of data transfers that must be carried out between the GPUs. The first type of data movement happens on each 2D (yz or xz or xy) plane, for the purpose of communicating results of the halo computation (i.e., on mesh points that lie immediately beside a neighbor) from one subdomain to another. Consider for example the partitioning shown in figure 5.9, which can be shared by sub-sweeps in the x and z-directions. Then parallel computing on each yz-plane (inside a_x-directional sub-sweep) will require one subdomain to send (and receive) n_z point values to (and from) each of its two neighbors. When updating each xy-plane (inside a z-directional sub-sweep), n_x point values are exchanged between a pair of neighboring subdomains. It should be noted that this halo-induced type of communication happens once per 2D plane.

The second type of data transfer is due to the need of switching the grid partitioning between some of the sub-sweeps. For example, the partitioning shown in figure 5.9 cannot be used for sub-sweeps along the y-direction. In connection with switching the grid partitioning, a 3D volumetric data shuffle among all GPUs is necessary. A concrete example of data shuffle involving four GPUs is shown in figure 5.10. There, each GPU can keep one fourth of its data, but has to exchange one fourth of its data wdith each of the other three GPUs. This volumetric data shuffle happens only twice per sweep (every six sub-sweeps). In other words, the

Listing 5.1: Basic

```
for (i=2; i<=nx; i++)
{
 cudaSetDevice(0);
 compute-kernel: update all points on GPU0 part of a yz-plane
 pack-kernel: fill a buffer containing halo values needed by GPU1

 cudaSetDevice(1);
 compute-kernel: update all points on GPU1 part of a yz-plane
 pack-kernel: fill a buffer containing halo values needed by GPU0

 cudaMemcpy (GPU0 buffer to host buffer H0)
 cudaMemcpy (GPU1 buffer to host buffer H1)
 cudaMemcpy (host buffer H0 to GPU1 buffer)
 cudaMemcpy (host buffer H1 to GPU0 buffer)

 cudaSetDevice(0);
 unpack-kernel: handle the incoming H1 data from host
 cudaSetDevice(1);
 unpack-kernel: handle the incoming H0 data from host
}
```

second type of communication is considerably less frequent than the first type of halo-induced communication.

5.5.5 CUDA implementations

5.5.5.1 Plain multi-GPU implementation

If there already exists a single-GPU CUDA implementation of a 3D sweep algorithm, it is a relatively simple programming task to write a plain implementation that uses multiple GPUs residing on the same compute node. Of the existing single-GPU code, its six kernels associated with the six different sub-sweeps can be reused by each GPU to work within its assigned subdomain of a 2D plane. Additional kernels have to be implemented to support the two types of data transfers: halo-induced communication and 3D volumetric data shuffle. The following pseudocode segment shows one sub-sweep in the positive x-direction for the case of using two GPUs:

There are two problems with the plain implementation above. First, due to the default synchronous CUDA stream on each GPU, the halo-induced communication will not start until all the mesh points on the subdomain 2D plane are updated. There is thus no possibility of hiding the overhead of this communication, as shown in figure 5.11. The second problem is the use of synchronous data copies (*cudaMemcpy* for both device-host and host-device transfers), meaning that only one data transfer can happen at a time. The same problem also applies to the second type of communication: 3D volumetric data shuffles (not shown in the above code

Listing 5.2: Improvement 1

```
for (i=2; i<=nx; i++)
{
 cudaSetDevice(0);
 halo-kernel using halo_stream(0)
 compute-kernel over interior points using compute_stream(0)
 cudaMemcpyAsync (GPU0->H0) using halo_stream(0)

 cudaSetDevice(1);
 halo-kernel using halo_stream(1)
 compute-kernel over interior points using compute_stream(1)
 cudaMemcpyAsync (GPU1->H1) using halo_stream(1)

 cudaStreamSynchronize halo_stream(0)
 cudaStreamSynchronize halo_stream(1)

 cudaSetDevice(0);
 cudaMemcpyAsync (H0->GPU1) using halo_stream(1)
 unpack-kernel using halo_stream(1)

 cudaSetDevice(1);
 cudaMemcpyAsync (H1->GPU0) using halo_stream(0)
 unpack-kernel using halo_stream(0)

 cudaSetDevice(0);
 cudaStreamSynchronize compute_stream(0);
 cudaSetDevice(1);
 cudaStreamSynchronize compute_stream(1);
}
```

segment).

5.5.5.2 Improvement 1

The key to hiding the overhead of halo-induced communications is to overlap this type of data transfers with computations of the interior points (i.e., non-halo points) on each GPU. For this purpose, every data-packing kernel from the plain implementation is extended (as a halo-kernel) to compute its line of "halo points" as well as packing. Moreover, each GPU adopts at least two CUDA streams, one being responsible for updating its interior mesh points, the other ("halo stream") for independently executing the data-packing kernels. Consequently, asynchronous device-host and host-device data transfers (by calling *cudaMemcpyAsync*) can be enabled by using the halo streams. The following pseudocode implements this improvement, and the effect can be seen in figure 5.12 for the case of two GPUs.

5.5.5.3 Improvement 2

The situation can be improved further. We note from figure 5.12 that the start of the kernels on GPU1 has a delay with respect to those on GPU0. This is because both GPUs are controlled by the same host CPU thread, which first initiates the kernels on GPU0 and then those on GPU1. This delay will become more severe when more GPUs are involved. To solve the above problems, we adopt the strategy of using multiple OpenMP threads on the CPU side, as proposed in [103]. That is, one controlling OpenMP thread is now in charge of each GPU. The entire code will thus be wrapped into an OpenMP parallel region, and the thread ID will dictate the responsibility of an OpenMP thread. The effect of this improvement is clearly visible in figure 5.13.

5.5.5.4 Improvement 3

As can be seen in figure 5.13, the combination of multiple CUDA streams and one controlling OpenMP thread per GPU can result in halo-induced data exchanges being carried out while computations on the interior mesh points proceed. The overhead of this type of communication can therefore be effectively hidden, even though the communication is relayed through the CPU host. For the second type of communication, i.e., 3D volumetric data shuffles between switches of the grid partitioning, relaying data via CPU is unnecessarily costly if there is hardware support for direct peer-to-peer (P2P) communication [104] between the GPUs. Specifically, to draw benefit from P2P and enable bi-directional data transfers, the *cudaMemcpyPeerAsync* function should be simultaneously called by the controlling OpenMP threads. Afterwards, the CUDA streams that execute the asynchronous P2P communication must be properly synchronized, via *cudaStreamSynchronize* called by the multiple controlling CPU threads, to make sure that the shuffled data has arrived. Finally, we remark that the above three improvements were first discussed in [109]. However, the multi-GPU implementations used in this paper have made substantial adjustments (and corrections) of those in [109].

5.5.6 Experiments and measurements

5.5.6.1 Hardware platforms

We tested our multi-GPU implementations on two GPU clusters, *Erik* [110] and *Zorn* [111], for running 3D simulations of anisotropic front propagation. In particular, one 4-GPU node on the Erik cluster was used, where each GPU is of type NVIDIA Tesla K20m. The CPU host has dual-socket 8-core Intel Xeon E5-2650 2.0 GHz processors. It is important to notice that the four GPUs are organized as two PCIe "islands" meaning that the GPU0-GPU1 and GPU2-GPU3 pairs have a direct PCIe connection in between, whereas across-pair traffic is subject to a slower speed. On the Zorn cluster, we used one of its 4-GPU nodes where each GPU is of type Tesla C2050 and the CPU host consists of dual-socket 4-core Intel Xeon E5620

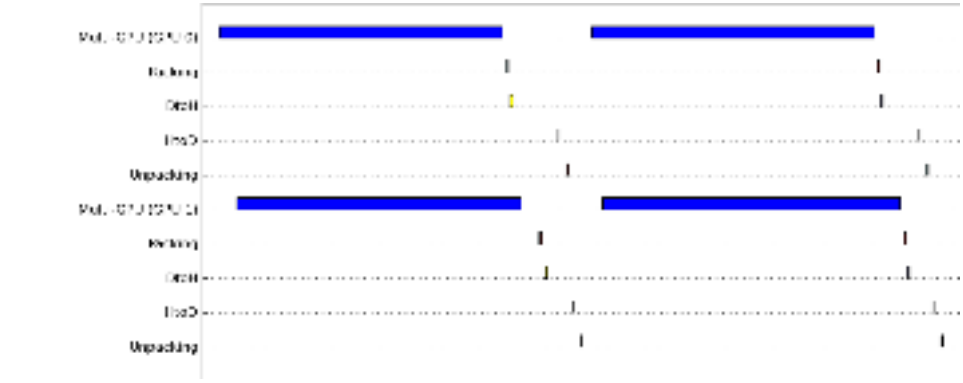


Figure 5.11: Plain 2-GPU implementation: the default synchronous CUDA stream per GPU

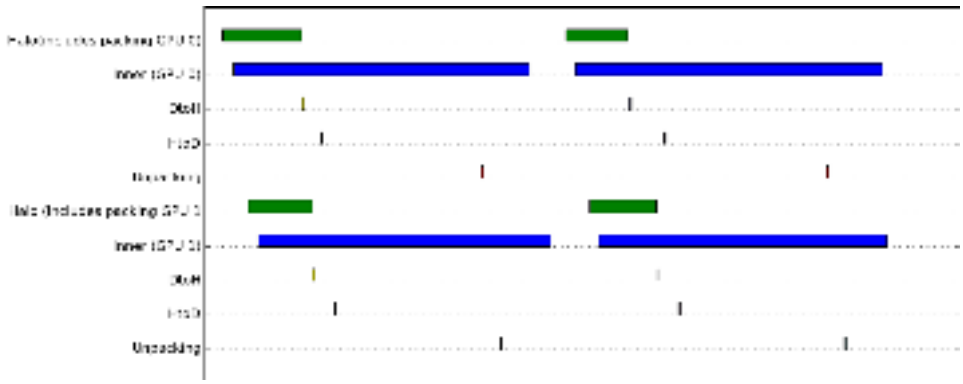


Figure 5.12: Improved 2-GPU implementation version 1: two CUDA streams per GPU

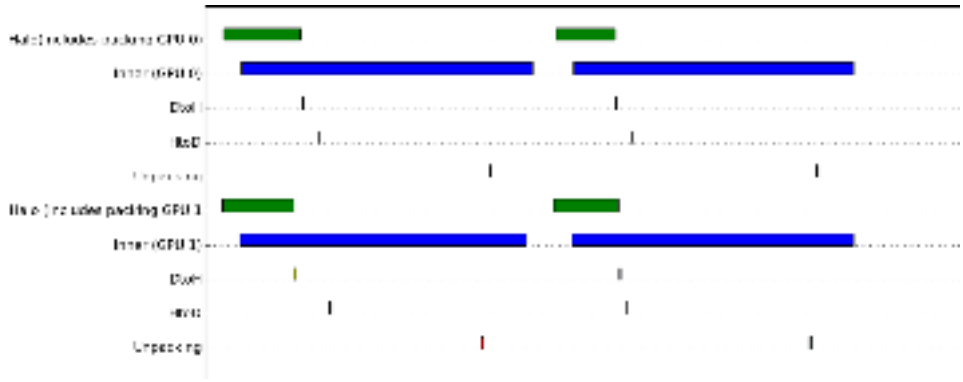


Figure 5.13: Improved 2-GPU implementation version 2: two CUDA streams and one OpenMP thread per GPU

2.4 GHz processors. The four GPUs are also organized as two pairs with intra-pair PCIe connection. CUDA v5.5 was used on both platforms.

5.5.6.2 An example of geological folding

To compare with the single-GPU implementation from [105], we have chosen to simulate the same example of geological folding. More specifically, $F = 1.1$, $\psi = 1.0$ and $\mathbf{a}$ = (-0.34, 0.4, 0.7) were used in 5.1. The 3D spatial domain has length 10 in all three directions. The initial surface Γ_0 is shown in the left plot of figure 5.14, whereas the simulation result is depicted in the right plot. (The numerical results from the multi-GPU implementations were all verified by those produced by the original single-GPU implementation.)

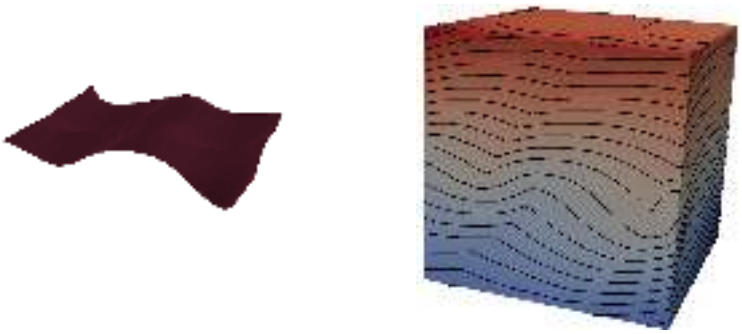


Figure 5.14: The initial surface Γ_0 (left plot) and the simulation result of (1) after running 8 sweeps

5.5.6.3 Time measurements

Table 1 summarizes the time measurements that were obtained on the Erik and Zorn systems. Two problem sizes were tested: $n_x = n_y = n_z = 512$ and $n_x = n_y = n_z = 640$. Each simulation ran 8 sweeps, i.e., 48 sub-sweeps in total. All computations used double precision. Recall from Section 4 that the first improvement to the plain multi-GPU implementation is to let each GPU use multiple CUDA streams. That is, the number of CUDA streams per GPU equals the

number of neighbors plus one. Halo computations are thus carried out as early as possible, and the data transfers (type 1 communication) between neighboring GPUs can be carried out while computations on the interior mesh points proceed. As can be seen in Table 1, the benefit of the first improvement is more obvious for the 4-GPU cases. Likely, the second improvement (assigning one controlling OpenMP thread per GPU) also has a clearer advantage for the 4-GPU cases. For the third improvement, using P2P data communication for the 3D volumetric data shuffles (type 2 communication) instead of relaying data via the CPU, the benefit is more profound for the 2-GPU cases. This is due to the fact that GPU0 and

Grid size	$n_x = n_y = n_z = 512$		$n_x = n_y = n_z = 640$	
	Time on Erik	Time on Zorn	Time on Erik	Time on Zorn
Single-GPU	31.71	64.95	59.54	123.34
2-GPU plain impl.	24.75	44.50	45.05	82.14
2-GPU improved v1	24.35	41.40	44.73	81.36
2-GPU improved v2	22.39	40.38	39.58	75.42
2-GPU improved v3	19.37	37.40	35.34	69.64
4-GPU plain impl.	23.28	30.77	41.03	54.61
4-GPU improved v1	20.68	28.54	38.65	51.56
4-GPU improved v2	14.31	25.54	25.18	46.83
4-GPU improved v3	12.48	23.57	21.91	43.02

Table 1: Time measurements (in seconds) of running eight sweeps of various multi-GPU implementations.

GPU1 have a direct PCIe connection that provides hardware support for P2P. When four GPUs are used, although GPU2 and GPU3 also form a pair with direct PCIe connection, data transfers across the two pairs (e.g. GPU0-GPU2) still have to relay through the CPU, thus not fully enjoying the performance benefit of P2P data communication.

5.5.7 Conclusions

In comparison with single-GPU implementations of parallel 3D sweeping algorithms, the use of multiple GPUs introduces the complicating issue of having to switch between two grid par- titionings and the resulting 3D volumetric data shuffles among all the GPUs. These come on top of the conventional halo-induced data exchanges between neighboring GPUs. In other words, parallelizing a 3D sweeping algorithm is more difficult than parallelizing a regular 3D finite difference method. The achievable parallel efficiency will consequently be lower due to the costly 3D volumetric data shuffles. Nevertheless, our time measurements have shown that with a proper use of multiple multiple CUDA streams per GPU, in combination with adopting one controlling OpenMP thread per GPU and P2P data communication offered by NVIDIA GPUDirect, we can secure satisfactory parallel efficiency. This means that using multiple GPUs for performing parallel 3D sweeps is a viable technical solution, especially considering the ben- efit of aggregating the device memory of the GPUs to solve larger problems that exceed the memory limit of a single GPU.

As future work, the current work can be extended to the scenario of multiple compute nodes, each with one or several GPUs. Asynchronous CUDA memory copies have to be replaced with suitable non-blocking MPI calls. Then, really huge-scale simulations can be made possible.

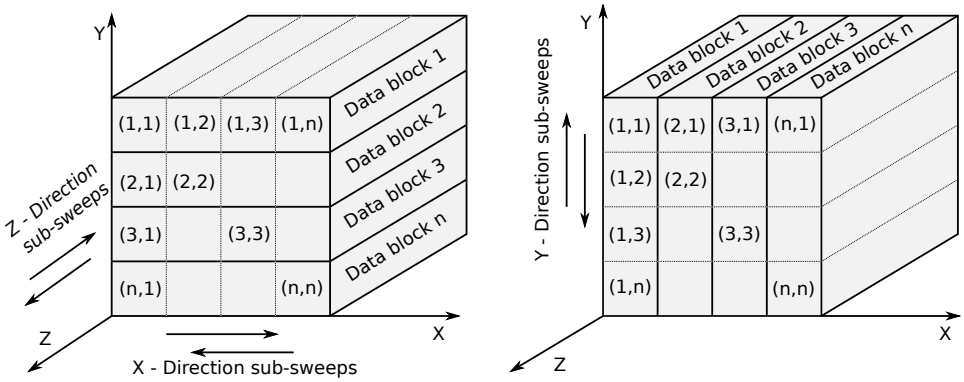


Figure 5.15: Data re-partition for the Y direction sub-sweeps.

5.5.8 Future work

A natural future extension is to investigate the FEniCS-HPC on GPU architectures, for example with new GPU sparse linear algebra backends, and with the Omega_h [112]library for general parallel mesh operations.

5.6 Towards HPC-embedded; case study-Kalray and message-passing on NoC

Today one of the most important challenges in HPC is the development of computers with low power consumption. In this context, recently, new embedded many-core systems have emerged. One of them is Kalray. Unlike other many-core architectures, Kalray is not a co-processor (self-hosted). One interesting feature of the Kalray architecture is the Network on Chip (NoC) connection. Habitually, the communication in many-core architectures is carried out via shared memory. However, in Kalray, the communication among processing elements can also be via Message-Passing on the NoC. One of the main motivations of this work is to present the main constraints to deal with the Kalray architecture. In particular, we focused on memory management and communication. We assess the use of NoC and shared memory on Kalray. Unlike shared memory, the implementation of Message-Passing on NoC is not transparent from programmer point of view. The synchronization among processing elements and NoC is another of the challenges to deal with in the Karlay processor. Although the synchronization using Message-Passing is more complex and consuming time than using shared memory, we obtain an overall speedup close to 6 when using Message-Passing on NoC with respect to the use of shared memory. Additionally, we have measured the power consumption of both approaches. Despite being faster, the use of NoC presents a higher power consumption with respect to the approach that exploits shared memory. This ad-

ditional consumption in Watts is about a 50%. However, the reduction in time by using NoC has an important impact on the overall power consumption as well.

5.6.1 Background

Advanced strategies for the efficient implementation of computationally intensive numerical methods have a strong interest in industrial and academic community. In the last decade, we have lived a spectacular growth in the use of many-core architectures for HPC applications [113–117]. However, the appearance of other (low-power consumption) embedded many-core architectures such as Kalray [118] has created new challenges and opportunities for performance optimization in multiple applications. In this work, we have explored some of these new opportunities towards supercomputing on a chip era.

Kalray integrates its own OS and is not in need of a co-processor as in the case of other many-core processors [118, 119]. In Karlay, highly expensive memory transfers from host main memory to co-processor memory are not necessary, as in other architectures, such as NVIDIA GPUs [120] or Inel MIC [121]. Besides, this architecture offers the possibility to communicate each of the processing elements via a Network on Chip (NoC) connection composed by links and routers [118, 119]. Kalray has been previously used for video encoding and Monte Carlo applications [7]. However, these works lack information of how to implement these applications and what are the most efficient programming strategies and architectonic features to deal with our embedded processor. The NoCs have been recently used as a level in-between the computing cores and shared memory [122–124]. The NoCs in these systems can be configurable, depending on the particular needs of the applications. However, the NoC in Kalray is completely different. In Kalray, there are two different and independent inter-connectors, one bus which connects each of the processing elements to shared memory and one NoC, which connects the different processing elements (clusters) among them.

We have chosen as a test case a widely known and extended problem, which is Jacobi method [125]. The main motivation of this work is twofold. While, on the one hand, this work presents the main challenges to deal with the Kalray architecture. On the other hand, we present two different approaches to implement the communication among the different processing elements of our Kalray processor, one based on using shared memory and other based on using a Network on Chip, which works as interconnection among the set of processing cores. We detail and analyze deeply each of the approaches, presenting theirs advantages and disadvantages. Moreover, we include measurements for power consumption in both approaches.

This section is structured as follows. Subsection 5.6.2 briefly introduces the main features of the architecture at hand, Kalray. Then, we detail the techniques performed for an efficient implementation of the Jacobi method on Kalray processor in Subsection 5.6.3. Finally, In Section 5.6.4, it is carried out the performance analysis of the proposed techniques in terms of consuming time, speed-up, and

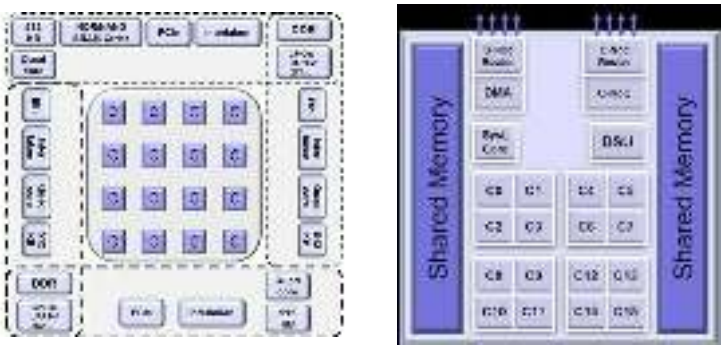


Figure 5.16: Kalray MPPA many-core (left) and compute cluster (righ) architecture [7]

power consumption. At the end of this work in subsection 5.6.5, we outline some conclusions.

5.6.2 Kalray Arhietecture

Kalray architecture [7] is an embedded many-core processor. It integrates 288 cores on a single 28 nm CMOS chip with a low power consumption per operation. We have 256 cores divided into 16 clusters, which are composed by 16+1 cores each. 4 quad-core I/O subsystems (1 core per cluster) are located at the periphery of the processing array (Figure 5.16-left). They are used as a DDR controller for accessing up to 64GB of external DDR3-1600. These subsystems control a 8-lane Gen3 PCI Express for a total peak throughput of 16GB/s full duplex. The 16 compute clusters and the 4 I/O subsystems are connected by two explicitly addressed Network on Chip (NoC) with bi-directional links, one for data and the other for control [7, 126]. NoC traffic does not interfere with the memory buses of the underlying I/O subsystem or compute cluster. The NoC is implemented following a 2-D torus topology.

The compute cluster (Figure 5.16 right) is the basic processing unit of our architecture [7]. Each cluster contains 16 processing cores (C0, C1, C2, . . . , C15 in Figure 5.16-right) and one resource management (Syst. Core in Figure 5.16-right) core, a shared memory, a direct memory access (DMA) controller, a Debug & System Unit (DSU), and two routers, one for data (D-NoC) and one for control (C-NoC). The DMA is responsible for transfer data among shared and the NoC with a total throughput of 3.2GB/s in full duplex. The shared memory compromises 2MB organized in 16 parallel banks, and with a bandwidth of 38.4 GB/s. The DSU supports the debug and diagnosis of the compute cluster.

Each processing or resource management core is a 5-way VLIW processor with two arithmetic and logic units, a multiply-accumulate & floating point unit, a load/store unit, and a branch & control unit [7]. It enables up to 800MFLOPS at 400MHz, which supposes almost 13 GFLOPS per cluster and almost 205 GFLOPS

in total by using the 16 clusters. These five execution units are connected to a shared register file, which allows 11 reads and 4 writes per cycle. Each core is connected to two (data & instruction) separate 2-way associate caches (8KB each).

Kalray provides a software development kit, a GNU C/C++ & GDB development tool for compilation, and debugging. Two programming models are currently supported. A high level programming model based on data-flow C language called $\sum C$ [127], where programmers do not care about communication, only data dependencies must be expressed. The other programming model supported is a POSIX-Level programming model [118, 119]. It distributes on I/O subsystems the sub-processes to be executed on the compute clusters and pass arguments through the traditional *argc*, *argv*, and *environ* variables. Inside compute clusters, classic shared memory programming models such as POSIX threads or OpenMP pragmas are supported to exploit more than one processing core. Specific IPC takes advantage of the NoC connection. Unlike $\sum C$, the POSIX-Level programming model presents more important challenges from programmer side, however, it allows us to have more control over hardware and optimize both communication and computation. In the present work, the authors have followed the programming model based on POSIX.

Algorithm 1 Jacobi OpenMP Algorithm.

1: *jacobi(A, Anew, N X, N Y)*
2: *float err;*
3: *#pragma omp parallel for*
4: **for** *int* $i = 1 \rightarrow NY - 1$ **do**
5: **for** *intj* $= 1 \rightarrow NX - 1$ **do**
6: $Anew[i * NX + j] = 0.25 * (A[i * NX + (j - 1)] + A[i * NX + (j + 1)] + A[(i-1) * NX + j] + A[(i+1) * NX + j]); err = maxf(err, fabs(Anew[i * NX + j] - A[i * NX + j]))$;
7: **end for**
8: **end for**
9: *#pragma omp parallel for*
10: **for** *inti* $= 1 \rightarrow NY - 1$ **do**
11: **for** *intj* $= 1 \rightarrow NX - 1$ **do**
12: **end for**
13: **end for**

5.6.3 Jacobi Method Implementation on Kalray

We have chosen as a test case the Jacobi method [125]. This is a good example, which allows us to study and evaluate different strategies for communication. The parallelization is implemented following a coarse-grained distribution of (adjacent) rows across all cores. This implementation is relatively straightforward using a

few OpenMP pragmas on the loops that iterate over the rows of our matrix (see Algorithm 1).

One of the most important challenges in Kalray is communication and memory management. To address the particular features of Kalray architecture, we use the Operating System called *NodeOs* [118], provided by Kalray. *NodeOs* implements the Asymmetric Multi-Processing (AMP) model. AMP takes advantage of the asymmetry found in the clusters between the Resource Management Core (RMC) and the Processing Element Cores (PEC). RMC runs the operating system (kernel and NoC routines) on the set of RM (single-core). PEC is dedicated to run user threads, one thread per PEC. PEC can also call functions, such as *syscall*, that need OS support, which is received and compute by RMC. When a PEC executes a *syscall* call, it sends an event, and it is locked until it receives an event from the RMC. This process is necessary to know that the *syscall* has been processed. Data and parameters are exchanged using shared memory. We have two codes, one executed by RMC (IO code) and other (*cluster* code) executed by PECs. The work is distributed following a master/slave model that is well suited to Kalray architecture. The IO code is the master. It is in charge of launching the code and sending data to be computed by slaves. Finally, they wait for the final results. Otherwise, the *cluster* code are the slaves. They wait for data to be computed and send results to IO cluster.

The POSIX-Level programming model of Kalray (*NodeOs*) allows us to implement communication among different clusters in two different ways. While shared memory (accessible by all clusters) is used for the communication in the first approach (SM), in the second approach (*NoC*), we use channels (links) and routers. For the sake of clarifying, we include several algorithms in which we detail the main steps of each of the approaches. Algorithms 2 and 3 illustrate the IO and *cluster* pseudocodes for the SM approach and Algorithms 4 and 5 for the *NoC* approach respectively.

The communication is implemented by using some specific objects and functions provided by *NodeOs*. Next, we explain each of these objects and functions. The transfers from/to global/local memory are implemented via *portals*. These portals must be initialized using specific paths (one path per cluster) as *A_portal* in Algorithm 2. Then, they must be opened (*mppa_open*) and synchronized (*mppa_ioctl*) before transferring (*mppa_pwrite* in Algorithm 2 and *mppa_aio_read* in Algorithm 3) data from/to global/local memory. The slaves are launched from master via *mppa_spawn* which include parameters and name of the function/s to be computed by cluster/s.

The communication among cluster via links (*NoC*) is implemented by using of *channel*. Similar to the use of *portals*, *channels* must be initialized using one path per channel (see *C0_to_C1 channel* in Algorithm 2).

On the other hand, the synchronization is implemented by using of *sync*. They are used to guarantee that some resources are ready to be used or cluster are ready to start computing (for instance, see *mppa_ioctl* in Algorithm 2,3,4 and 5).

In order to minimize the number of transfers among main and local memory

Algorithm 2 Shared Memory I/O pseudocode.

```
1: const char * cluster executable = "mainCLUSTER";
2: static f loat A[SIZE]; static f loat Anew[SIZE];
3: int mainIO(int argc , char * argv[] )
4: long long dummy = 0; long long match = -(1 « CLUSTER_COUNT);
5: const char * root sync = "/mppa/sync/128 : 1";
6: const char * A portal = "/mppa/portal/"CLUSTER_RANGE" : 1";
7: const char * Anew portal = "/mppa/portal/128 : 3";
8: //–OPENING FILES–//
9: int root_sync_fd = mppa_open(root_sync, O_RDONLY );
10: int A_portal_fd = mppa_open(A_portal, O_W RONLY );
11: int A_new_portal_fd = mppa_open(Anew_portal, O_RDONLY );
12: //–PREPARE_FOR_RESULT–//
13: status| = mppa_ioctl(root_sync_fd, MPPA_RX_SET_ MATCH, match);
14: mppa_aicob_t Anew_portal_aiocb[1] = {MPPA_AIOCB_INITIALIZER
    (Anew_portal_fd, Anew, sizeof (Anew[0]) * SIZE)};
15: mppa_aiocb_set_trigger(Anew_portal_aiocb, CLUSTER_COUNT);
16: status| = mppa_aio_read(Anew_portal_aiocb);
17: //–LAUNCHING SLAVES–//
18: char arg0[10], arg1[10];
19: const char * argv[] = arg0, root_sync, A_portal, Anew_portal, 0;
20: for int rank = 1 → CLUSTER_COUNT do
21:     sprintf (arg0, "%d", rank);
22:     status| = mppa_spawn(rank, NULL, cluster_executable, argv, 0);
23: end for
24: //Wait for the cluster portals to be initialized.
25: status| = mppa_read(root_sync_fd,& dummy, sizeof(dummy));
26: // Distribute slices of array A over the clusters.
27: for int rank = 0 → CLUSTER_COUNT do
28:     status| = mppa_ioctl(A_portal_fd, MPPA_TX_SET_RX_RANK,
    rank);
29:     status| = mppa_pwrite(A_ portal_fd, (A + rank * SIZE_ LOCAL) -
    (NX_ LOCAL * 2),sizeof (float) * SIZE_LOCAL, 0);
30: end for
31: // Wait for the cluster contributions to arrive in array |Anew|.
32: status| = mppa_aio_ wait(Anew_portal_aiocb);
33: return status < 0;
```

(*SM* approach) as well as among clusters through links (*NoC* approach), the matrix is divided into rectangular sub-blocks (Figures 5.17 and 5.18). In particular, the distribution of the workload and communication implemented in the *NoC* approach avoid multi-level routing, connecting each of the cluster with its adjacent clusters via a direct link.

Although the ghost cell strategy is usually used for communication in distributed memory systems [128], we have used this strategy in Kalray processor to avoid race conditions among each of the sub-blocks assigned to each clusters. The use of ghost cells consists of replicating the borders of all immediate neighbor blocks. These ghost cells are not updated locally but provide stencil values when updating the borders of local blocks. Every ghost cell is a duplicate of a piece of memory located in neighbors nodes. To clarify, Figures 5.17 and 5.18 illustrate a simple scheme for our interpretation of the ghost cell strategy applied to both approaches, *SM* and *NoC*, respectively.

Algorithm 3 Shared Memory CLUSTER Pseudocode.

```
1: int mainCLUSTER(int argc, chjar *argv[])
2: int i, j, k, status, rank = atoi(argv[0]);
3: const char * root_sync = argv[1],*A_portal = argv[2],*Anew_portal = argv[3];
4: float A[SIZE_LOCAL], Anew[SIZE_LOCAL]; long long slice_offset;
5: slice_offset = sizeof(float)*(CHUNK * NX _LOCAL+((rank-1)*(CHUNK-
   1)*NX _LOCAL));
6: Each clster contributes a different bit to the root_sync mask
7: long long mask = (long long)1 rank;
8: //–OPENING_PORTAL–//
9: int root_sync_fd = mppa_open(root_sync, O_W RONLY);
10: int A_portal_fd = mppa_open(A_portal, O_RDONLY);
11: int Anew_portal_fd = mppa_open(Anew_portal, O_W RONLY);
12: //–PREPARE_FOR_INPUT–//
13: mppa_aiocb_t A_portal_aiocb[1] = MPPA_AIOCB_INITIALIZER(A_portal_fd,
   A, sizeof(A));
14: status| = mppa_aio_read(A_portal_aiocb);
15: –UNLOCK_MASTER–//
16: status| = mppa write(root_sync_fd, & mask, sizeof (mask));
17: // Wait for notification of remote writes to local arrays |A|.
18: status| = mppa_aio wait(A_portal_aiocb);
19: //–JACOBIANCOMPUTE–//
20: jacobi(A, Anew, NX _LOCAL, NY _LOCAL);
21: //Contribute back local array Anew into the portal of master array
   Anew.
22: status| = mppa pwrite(Anew_portal_fd, & Anew[NX _LOCAL], sizeof
   (Anew) - sizeof (float) âĹŮ 2 âĹŮ NX _LOCAL, slice_offset);
23: mppa exit((status < 0)); return 0;
```

Figure 5.17 graphically illustrates the strategy followed by the *SM* approach. It consists of dividing the matrix into equal blocks which are sent from main memory to local memory. To avoid race condition, each of the blocks includes 2 additional rows (gray and white rows in Figure 5.17) which correspond to the upper and lower adjacent rows of the block. These additional rows work as ghost-cell, which are only used in local memory. The blocks transferred from local memory to global memory (Figure 5.17-right) do not include these additional rows (ghost rows).

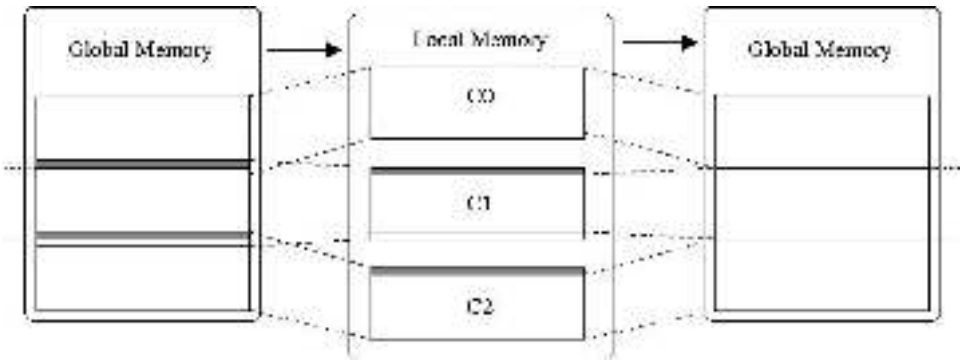


Figure 5.17: Master (Global Memory) ↔ Slave (Local Memory) Communication.

Otherwise the communication among global and local memory is not necessary in the *NoC* approach. The master (IO code) is only used for synchronizing. The synchronization is necessary at the beginning and at the end of each *Master* code. I/O core and the rest of cores in each of the clusters must be also synchronized. In particular the synchronization between IO core and computing cores ($I/O- > C1$ *sync* section in Algorithm 5) is necessary to guarantee that there are no cluster reading into channels before the corresponding cluster has opened the channel. After computing the Jacobi method in each of the clusters, some rows of the local blocks must be transferred to/from adjacent clusters. The first row computed (white upper row C1 in Figure 5.18) must be transferred to the upper adjacent cluster (C0) to be stored in the last row. Also, the last row computed (gray lower row C1 in Figure 5.18) must be transfered to the lower adjacent cluster (C2) to be stored in the first row. This pattern must be carried out in all clusters except the first and last clusters where a lower number of data-transfers is necessary.

5.6.4 Performance Study

In this section, we analyze deeply both approaches, *SM* and *NoC*, focusing on communication, synchronization and computing separately. In order to find/focus on the performance of both approaches, we have used a relatively small problem which can be fully stored in local memory.

Algorithm 4 NoC I/O pseudocode.

```
1: const char * global_sync = "/mppa/sync/128:1";
2: const char * IO_to_CO_sync = "/mppa/sync/0:2";...
3: const char * C0_to_C1_channel = "/mppa/channel/1:1/0:1";...
4: static const char *exe[CLUSTER_COUNT] = {"mainCLUSTER0","mainCLUSTER1",...};
5: int mainIO(int argc, const char * argv[])
6: //Global sync
7: int ret, global_sync_fd = mppa_open(global_sync,O_RDONLY
8: long long match = -1 « CLUSTER_COUNT
9: mppa_ioctl(global_sync_fd, MPPA_RT_SET_MATCH, match);
10: //–IO_TO_C # _SYNC–//
11: int IO_to_C0_sync_fd = mppa_open(IO_to_C0_sync, O_W RONLY);
12: int IO_to_C1_sync_fd = mppa_open(IO_to_C1_sync, O_W RONLY);
13: //–LAUNCHING_SLAVES–//
14: for inti =→ CLUSTER_COUNT do
15:     mppa_spawn(i,NULL,exe[i],argv,0);
16: end for
17: //Wait for other clusters to be ready.
18: mppa_read(global_sync_fd, NULL, 8);
19: Write into I/O -> C # sync to unlock C # cluster.
20: mask = 1; mppa_write(IO_to_C0_sync_fd, &mask, sizeof(mask));
21: mppa_write(IO_to_C1_sync_fd, &mask, sizeof(mask));...
22: //–WAITING TO THE END OF CLUSTERS EXECUTION–//
23: for inti = 0 → CLUSTER_COUNT do
24:     red = mppa_waitpid(i,&status,0);mppa_exit(ret);
25: end for
```

Next we present the commands used to compile and launch both approaches:
Compiling lines:
$k1 - gcc\ -O3\ -std = c99\ -mos = rtems\ io.c\ -o\ io\_app\ -lmppaipc$
$k1 - gcc\ -O3\ -std = c99\ -fopenmp\ -mos = nodeos\ cluster.c\ -o\ cluster\ -lmppaipc$
$k1 - create - multibinary\ - -\ cluster\ cluster\ - -\ boot = io\_app\ -T\ multibin$
Launching line:
$k1 - jtag - runner\ - -\ multibinary\ multibin\ - -\ exec - multibin = IODDR0 :\ io\_app$

The communication among I/O and computing cores in the *NoC* approach is more complex and it is in need of a higher number of synchronizations. This causes a higher execution time with respect to the *SM* approach, being almost 2.5× bigger (Figures 5.19 and 5.20). Note that we use a different vertical scaling in each of the

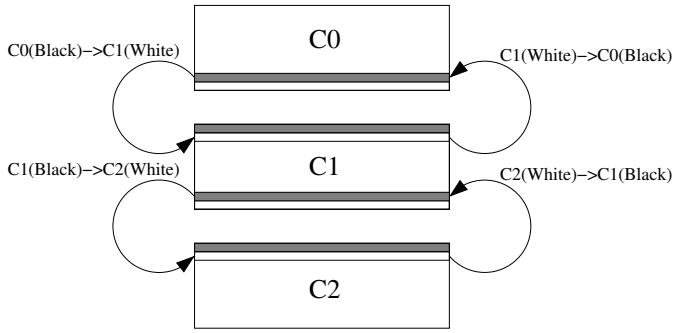


Figure 5.18: Pipeline (Bus) Communication.

graphics illustrated in Figures 5.19 and 5.20 .Despite of the overhead caused by a higher number of synchronizations, the use of the *NoC* interconnection makes the *NoC* approach (Figure 5.20) about 55× faster than the *SM* approach.

As expected the time consumed for computing the Jacobi method is equivalent in both approaches. The time consumed by synchronization, communication and computing in the *NoC* approach is more balanced than in the *SM* approach. This can be beneficial for future improvements, such as asynchronous communication.

Finally, we analyse the performance in terms of GFLOPS. First, we compute the theoretical FLOPS for the Jacobi computation. The variant used in this study performs six flops per update (Algorithm 1). Therefore, the theoretical FLOPS is equal to the elements of our matrix multiplied by six.

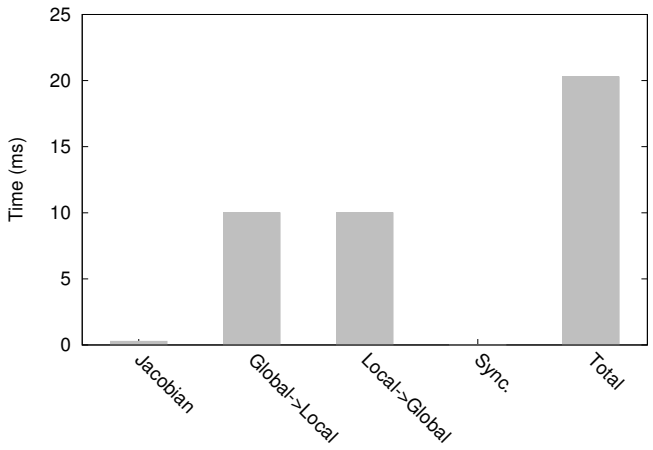


Figure 5.19: Time consumption for the *SM* approach.

Algorithm 5 *NoC* CLUSTER pseudocode

```
1: intmainCLUSTER1(intargc, char * argv[])
2: floatA[SIZE_LOCAL], Anew[SIZE_LOCAL];
3: //Openalltheresourcesneededfortransfers.
4: //Globalsync.
5: intglobal_sync_fd = mppa_open(global_sync, O_WRONLY);
6: //C1- > C2channel.
7: intchannel0_fd = mppa_open(C1_to_C2_channel, O_WRONLY);
8: //C2- > C1channel.
9: intchannel1_fd = mppa_open(C2_to_C1_channel, O_RDONLY);
10: //C1- > C0channel.
11: intchannel2_fd = mppa_open(C1_to_C0_channel, O_WRONLY);
12: //C0- > C1channel.
13: intchannel3_fd = mppa_open(C0_to_C1_channel, O_RDONLY);
14: //I/O - C1sync.
15: intIO_to_C1_sync_fd = mppa_open(IO_to_C1_sync, O_RDONLY);
16: longlongmatch = -(1 << 1/ * WesynconlywithI/Ocluster * /);
17: mppa_ioctl(IO_to_C1_sync_fd, MPPA_RX_SET_MATCH, match)
18: //WriteintoglobalsynctounlockI/Ocluster.
19: longlongmask = 1 << mppa_getpid();
20: mppa_write(global_sync_fd, &mask, sizeof(mask))
21: // - -WAIT_FOR_IO_TO_C1_SYNC - -//
22: mppa_read(IO_to_C1_sync_fd, NULL, 8);
23: // - -CLUSTERSCOMMUNICATION - -//
24: //Writedataforcluster0.
25: mppa_write(channel0_fd, &A[NX_LOCAL * (NY_LOCAL - 2)], sizeof(float) *
    NX_LOCAL);
26: //ReaddatafromC0.
27: mppa_read(channel1_fd, A, sizeof(float) * NX_LOCAL);
28: //ReaddatafromC2.
29: mppa_write(channel2_fd, &A[NX_LOCAL], sizeof(float) * NX_LOCAL);
30: //Writedataforcluster2.
31: mppa_read(channel3_fd, &A[NX_LOCAL * (NY_LOCAL - 1)], sizeof(float) *
    NX_LOCAL);
32: mppa_exit(0);
```

In order to evaluate the overhead of each of the strategies, first, we show the GFLOPS achieved by the Jacobi computation without the influence of the synchronization and communication (see *Jacobian* in Figure 5.21). It achieves almost the peak of performance of our platform (*GFLOPS-Peak* in Figure 5.21). The computation of the Jacobian method is exactly the same in both approaches (SM and *NoC*). Next, we include the overhead of the communication. Although both approaches present a fall in performance when taking into account the time consumed by the communication, the fall shown by the *NoC* approach is not so dramatic as the overhead suffered by the SM approach (Figure 5.21).

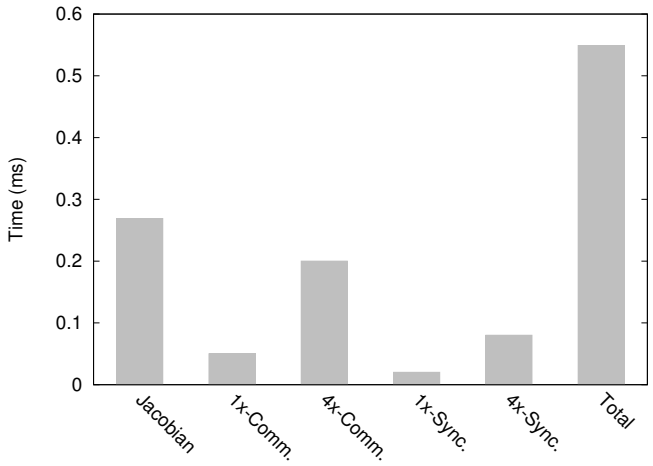


Figure 5.20: Time consumption for the *NoC* approach.

The software development kit provided by Kalray allow us to measure the power consumption of our applications. This is done via this command:

$k1 - power\ --\ k1 - jtag - runner\ --\ multibinary\ multibin\ --\ exec - multibin = IODDR0 : io\_app$

Executing our binary using *k1-power* we obtain the power achieved in terms of Watts. The average power achieved by the *NoC* approach is about 8.508W , while the SM approach achieves an average of $5.778W$ in every execution. This is almost a 50% more power when executing the *NoC* approach. However, the reduction in execution time obtained by the *NoC* approach has an important impact on the overall power consumed. Joules are computed by following the next expression:

$Joules = Watts \times Time$

obtaining an overall consumption about $0.0047J$ and $0.16J$ for the *NoC* and the *SM* approaches respectively. This is a 96% less of power consumed by the *NoC* approach.

5.6.5 Conclusions and Future Work

Embedded many-core architectures such as Kalray have emerged as a new HPC platform to deal with the problem of the excessive power consumption.

In this work, we have presented two different approaches to implement the communication among the processing elements of the Kalray architecture. Both approaches implement a ghost-cell strategy to avoid race conditions among the different blocks assigned to each of the processing elements (clusters). This strat-

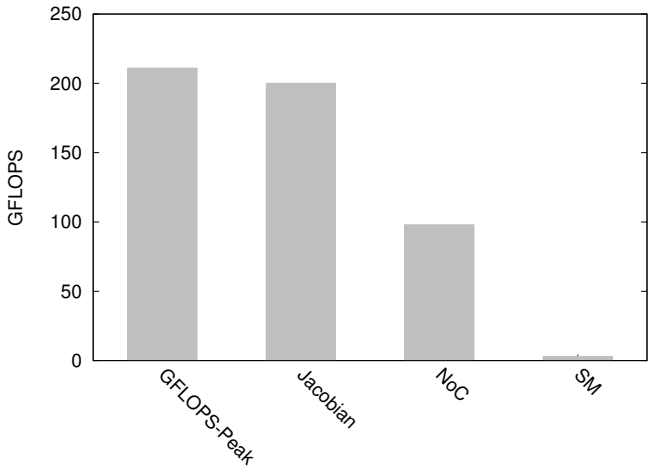


Figure 5.21: GFLOPS achieved by both approaches.

egy has been adapted to the particular features of our embedded processor and approaches, *SM* and *NoC*, to minimize the number of transfers.

Although the communication via shared memory is more habitual and easier to implement on many-core architectures, the particular features of the Kalray architecture, in particular the communication via Message- Passing on *NoC* connection, offers a much faster alternative. Although, the use of *NoC* consumes more power, the reduction in time makes this approach more efficient in terms of power consumption.

We plan to investigate other problems and more efficient strategies for memory management and data distribution, such as the overlapping of communication and computing via asynchronous transfers. In particular, the *NoC* approach could take advantage of the asynchronous communication as the time consumed by its major steps is balanced.

Chapter 6

Outcomes and future work

We have shown that with our DFS methodology and open-source Unicorn/FEniCS-HPC automated software framework, we can predict the stall of a realistic aircraft at realistic Reynolds number, which is considered by NASA as one of the grand challenges problems to be solved by 2030 [1]. We not only predict aerodynamic forces close to experimental results but also 10 times cheaper and faster when compared to existing CFD methodologies. Our work in this direction has been highlighted by NASA, a Fields Medalist former, KVA Royal Swedish Academy of Engineering Sciences and at the highest echelon of the aerodynamics industry. In order to promote this technology as an open-source scientific platform in the CFD community and industry, we have started an open-source spin-off called "Icarus Digital Math", supported by KTH Innovation, Vinnova and KVA Royal Swedish Academy of Engineering Sciences. Our aim is to transform the industry to Digital Math - science as open-source, which is verifiable and can be used by anyone at an affordable cost.

Our variable density approach does not yet include an adaptive algorithm, which means we do not have any posteriori error estimation. Including adaptivity would be beneficial in terms of computational cost and reliability. HarshLab dynamics in the multiphase setting can be used for the further related application. And also, it is good to focus on Cloud technology as well since DFS does not need so much computational power for smaller problems plus cloud platforms have the latest hardware (CPU and GPU) to be used, especially on the Amazon cloud platform.

One of the grand challenges in CFD to be solved by 2030, according to NASA is for CFD solvers to attain exascale computing using massively parallel and heterogeneous architectures. At present, FEniCS-HPC runs only on homogeneous architectures. This is due to a bottleneck of data transfer in FEM between CPU and GPU, studies with FEniCS on early stage GPU architectures have not been able to attain competitive speedup. In the past few years, GPU technology has attained tremendous technological advancement in terms of a number of cores, data transfer (through NVlink), and unified memory. FEniCS-HPC on heterogeneous

architectures would be beneficial, especially since most new supercomputers have a heterogeneous architecture.

In our adaptive methodology in DFS and FEniCS-HPC, we refine the mesh in parallel based on goal-oriented duality-based error control. One additional key step in FEniCS-HPC is to allow parallel mesh coarsening and general mesh operations. We identified the Open Source library Omega_h as providing this key technology that would unlock general adaptivity.

Bibliography

[1] J. Slotnick, A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie, and D. Mavriplis, "Cfd vision 2030 study: a path to revolutionary computational aerosciences," 2014.

[2] "Vtk." [Online]. Available: https://vtk.org/

[3] Visit. [Online]. Available: https://wci.llnl.gov/simulation/computer-codes/visit

[4] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[5] "Aerodynamics: Flow around the ahmed body." [Online]. Available: https://www.simscale.com/aerodynamics-flow-around-the-ahmed-body/

[6] "Ansys meshing." [Online]. Available: https://www.ansys.com/products/platform/ansys-meshing

[7] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2013, pp. 1–6.

[8] J. Hoffman, J. Jansson, and N. Jansson, "Fenics-hpc: Automated predictive high-performance finite element computing with applications in aerodynamics," *Proceedings of the 11th International Conference on Parallel Processing and Applied Mathematics, PPAM 2015. Lecture Notes in Computer Science*, 2015.

[9] J. Hoffman and C. Johnson, *Computational Turbulent Incompressible Flow*, ser. Applied Mathematics: Body and Soul. Springer, 2007, vol. 4.

[10] J. Hoffman, J. Jansson, R. V. de Abreu, N. C. Degirmenci, N. Jansson, K. Müller, M. Nazarov, and J. H. Spühler, "Unicorn: Parallel adaptive finite element simulation of turbulent flow and fluid-structure interaction for

deforming domains and complex geometry," *Comput. Fluids*, vol. 80, no. 0, pp. 310 – 319, 2013.

[11] J. Hoffman, J. Jansson, C. Degirmenci, N. Jansson, and M. Nazarov, *Unicorn: a Unified Continuum Mechanics Solver*. Springer, 2012, ch. 18.

[12] A. Logg, K.-A. Mardal, G. N. Wells *et al.*, *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[13] FEniCS, "Fenics project," *http://www.fenicsproject.org*, 2003.

[14] J. Hoffman, J. Jansson, and M. Stöckli, "Unified continuum modeling of fluid-structure interaction," *Mathematical Models and Methods in Applied Sciences*, 2011.

[15] A. Logg, G. N. Wells, and J. Hake, "Dolfin: A c++/python finite element library," in *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012, pp. 173–225.

[16] R. C. Kirby, "Algorithm 839: Fiat, a new paradigm for computing finite element basis functions," *ACM Transactions on Mathematical Software (TOMS)*, 2004.

[17] R. C. Kirby and A. Logg, "A compiler for variational forms," *ACM Transactions on Mathematical Software*, vol. 32, no. 3, pp. 417–444, 2006.

[18] N. Jansson, J. Hoffman, and J. Jansson, "Framework for Massively Parallel Adaptive Finite Element Computational Fluid Dynamics on Tetrahedral Meshes," *SIAM J. Sci. Comput.*, vol. 34, no. 1, pp. C24–C41, 2012.

[19] J. Hoffman, J. Jansson, R. V. de Abreu, N. C. Degirmenci, N. Jansson, K. Müller, M. Nazarov, and J. H. Spühler, "Unicorn: Parallel adaptive finite element simulation of turbulent flow and fluid-structure interaction for deforming domains and complex geometry," *Comput. Fluids*, vol. 80, no. 0, pp. 310 – 319, 2013.

[20] J. Hoffman and C. Johnson, *Computational Turbulent Incompressible Flow*, ser. Applied Mathematics: Body and Soul. Springer, 2007, vol. 4.

[21] N. Jansson, "High performance adaptive finite element methods for turbulent fluid flow," Ph.D. dissertation, KTH Royal Institute of Technology, 2011.

[22] G. Houzeaux, M. Vázquez, R. Aubry, and J. Cela, "A massively parallel fractional step solver for incompressible flows," *Journal of Computational Physics*, vol. 228, no. 17, pp. 6316–6332, 2009.

[23] J. Hoffman and C. Johnson, *Computational Turbulent Incompressible Flow: Applied Mathematics Body and Soul Vol 4*. Springer-Verlag Publishing, 2006.

[24] J. Hoffman, J. Jansson, N. Jansson, R. Vilela De Abreu, and C. Johnson, "Computability and adaptivity in cfd. encyclopedia of computational mechanics, stein, e., de horz, r. and hughes, tjr eds," 2016.

[25] J. Hoffman and C. Johnson, *Adaptive finite element methods for incompressible fluid flow.* Heidelberg: Error Estimation and Solution Adaptive Discretization in Computational Fluid Dynamics (Ed. T. J. Barth and H. Deconinck), Lecture Notes in Computational Science and Engineering, Springer-Verlag Publishing, 2002, pp. 97–158.

[26] ——, "A new approach to computational turbulence modeling," *Comput. Methods Appl. Mech. Engrg.*, vol. 195, pp. 2865–2880, 2006.

[27] J. Hoffman, "Adaptive simulation of the turbulent flow past a sphere," *J. Fluid Mech.*, vol. 568, pp. 77–88, 2006.

[28] ——, "Efficient computation of mean drag for the subcritical flow past a circular cylinder using general galerkin g2," *Int. J. Numer. Meth. Fluids*, vol. 59(11), pp. 1241–1258, 2009.

[29] J. Hoffman and N. Jansson, *A computational study of turbulent flow separation for a circular cylinder using skin friction boundary conditions.* Ercoftac, series Vol.16, Springer, 2010.

[30] J. Hoffman and C. Johnson, "Resolution of d'alembert's paradox," *J. Math. Fluid Mech.*, Published Online First at www.springerlink.com: 10 December 2008.

[31] R. Vilela de Abreu, N. Jansson, and J. Hoffman, "Adaptive computation of aeroacoustic sources for a rudimentary landing gear," *Int. J. Numer. Meth. Fluids*, vol. 74, no. 6, pp. 406–421, 2014. [Online]. Available: http://dx.doi.org/10.1002/fld.3856

[32] P. Schlatter and R. Örlü, "Turbulent boundary layers at moderate reynolds numbers: inflow length and tripping effects," *Journal of Fluid Mechanics*, vol. 710, pp. 5–34, 2012.

[33] FEniCS-HPC, "Fenics-hpc project," *https://bitbucket.org/fenics-hpc/*.

[34] J. Hoffman and C. Johnson, *Computational turbulent incompressible flow: Applied mathematics: Body and soul 4.* Springer Science & Business Media, 2007, vol. 4.

[35] K. Kleefsman, G. Fekken, A. Veldman, B. Iwanowski, and B. Buchner, "A volume-of-fluid based simulation method for wave impact problems," *Journal of Computational Physics*, vol. 206, no. 1, pp. 363–393, 2005.

[36] X. Deng, M. Mao, G. Tu, H. Zhang, and Y. Zhang, "High-order and high accurate cfd methods and their applications for complex grid problems," *Communications in Computational Physics*, vol. 11, no. 4, pp. 1081–1102, 2012.

[37] J. A. Ekaterinaris, "High-order accurate, low numerical diffusion methods for aerodynamics," *Progress in Aerospace Sciences*, vol. 41, no. 3-4, pp. 192–300, 2005.

[38] Z. Wang, "High-order methods for the euler and navier–stokes equations on unstructured grids," *Progress in Aerospace Sciences*, vol. 43, no. 1-3, pp. 1–41, 2007.

[39] F. D. Witherden, A. M. Farrington, and P. E. Vincent, "Pyfr: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach," *Computer Physics Communications*, vol. 185, no. 11, pp. 3028–3040, 2014.

[40] A. Stuermer, "Unsteady cfd simulations of propeller installation effects," in *42nd AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit*, 2006, p. 4969.

[41] R. Mustak, M. H. U. Khan, M. M. Rahman, and M. Mashud, "Investigation of slipstreaming effect between a semi trailer-truck and a sedan car," *International Journal of Scientific & Engineering Research (IJSER)*, vol. 8, no. 2, pp. 676–678, 2017.

[42] L. Pii, E. Vanoli, F. Polidoro, S. Gautier, and A. Tabbal, "A full scale simulation of a high speed train for slipstream prediction," in *Transport Research Arena (TRA) 5th Conference: Transport Solutions from Research to DeploymentEuropean CommissionConference of European Directors of Roads (CEDR) European Road Transport Research Advisory Council (ERTRAC) WATERBORNEáţĂát¿European Rail Research Advisory Council (ERRAC) Institut Francais des Sciences et Technologies des Transports, de l'Aménagement et des Réseaux (IFSTTAR) Ministère de l'Écologie, du Développement Durable et de l'Énergie*, 2014.

[43] C. Baker, S. Dalley, T. Johnson, A. Quinn, and N. Wright, "The slipstream and wake of a high-speed train," *Proceedings of the Institution of Mechanical Engineers, Part F: Journal of Rail and Rapid Transit*, vol. 215, no. 2, pp. 83–99, 2001.

[44] F. D. Witherden and A. Jameson, "Future directions of computational fluid dynamics," in *23rd AIAA Computational Fluid Dynamics Conference*, 2017, p. 3791.

[45] H. Shan, L. Jiang, and C. Liu, "Direct numerical simulation of flow separation around a naca 0012 airfoil," *Computers and Fluids*, vol. 34, p. 1096ï¿Ř1114, 2005.

[46] P. Sagaut, *Large Eddy Simulation for Incompressible Flows (3rd Ed.)*. Springer-Verlag, Berlin, Heidelberg, New York, 2005.

[47] P. Moin and D. You, "Active control of flow separation over an airfoil using synthetic jets," *Journal of Fluids and Structures*, vol. 24, no. 8, pp. 1349–1357, 2008.

[48] L. Huang, P. G. Huang, and R. P. LeBeau, "Numerical study of blowing and suction control mechanism on naca 0012 airfoil," *AIAA Journal of aircraft*, vol. 41, no. 1, 2004.

[49] P. R. Spalart, "Detached-eddy simulation," *Annu Rev. Fluid Mech.*, vol. 41, pp. 181–202, 2009.

[50] U. Piomelli and E. Balaras, "Wall-layer models for large-eddy simulation," *Annu. Rev. Fluid Mech.*, vol. 34, pp. 349–374, 2002.

[51] C. P. Mellen, J. Frölich, and W. Rodi, "Lessons from lesfoil project on large-eddy simulation of flow around an airfoil," *AIAA journal*, vol. 41, pp. 573–581, 2003.

[52] J. Hoffman, "Computation of mean drag for bluff body problems using adaptive dns/les," *SIAM J. Sci. Comput.*, vol. 27(1), pp. 184–207, 2005.

[53] J. Hoffman, J. Jansson, N. Jansson, and R. V. D. Abreu, "Towards a parameter-free method for high reynolds number turbulent flow simulation based on adaptive finite element approximation," *Computer Methods in Applied Mechanics and Engineering*, vol. 288, no. 0, pp. 60 – 74, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045782514004836

[54] R. C. Kirby, *FIAT: Numerical Construction of Finite Element Basis Functions,*. Springer, 2012, ch. 13.

[55] A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, *FFC: the FEniCS Form Compiler*. Springer, 2012, ch. 11.

[56] J. Hoffman, J. Jansson, N. Jansson, and M. Nazarov, "Unicorn: A unified continuum mechanics solver," in *Automated Solutions of Differential Equations by the Finite Element Method*. Springer, 2011. [Online]. Available: http://www.fenicsproject.org/pub/documents/book/

[57] J. Hoffman, J. Jansson, N. Jansson, C. Johnson, and R. V. de Abreu, "Turbulent flow and fluid-structure interaction," in *Automated Solutions of Differential Equations by the Finite Element Method*. Springer, 2011. [Online]. Available: http://www.fenicsproject.org/pub/documents/book/

[58] C. Rumsey, "3^{rd} AIAA CFD High Lift Prediction Workshop (HiLiftPW-2) (http://hiliftpw.larc.nasa.gov/)," 2017. [Online]. Available: http://hiliftpw.larc.nasa.gov/

[59] J. C. Hunt, A. A. Wray, and P. Moin, "Eddies, streams, and convergence zones in turbulent flows," 1988.

[60] J. Hoffman, J. Jansson, R. V. de Abreu, N. C. Degirmenci, N. Jansson, K. Müller, M. Nazarov, and J. H. Spühler, "Unicorn: Parallel adaptive finite element simulation of turbulent flow and fluid–structure interaction for deforming domains and complex geometry," *Computers & Fluids*, vol. 80, pp. 310–319, 2013.

[61] J. Hoffman, J. Jansson, and N. Jansson, "Fenics-hpc: Automated predictive high-performance finite element computing with applications in aerodynamics," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2015, pp. 356–365.

[62] S. N. Jonkman, "Global perspectives on loss of human life caused by floods," *Natural hazards*, vol. 34, no. 2, pp. 151–175, 2005.

[63] The international disaster database. [Online]. Available: http://www.emdat.be/country{_}profile/index.html

[64] H.-M. Füssel, A. Jol *et al.*, "Climate change, impacts and vulnerability in europe 2012 an indicator-based report," 2012.

[65] A. Ezcurra, J. Areitio, and I. Herrero, "Relationships between cloud-to-ground lightning and surface rainfall during 1992–1996 in the spanish basque country area," *Atmospheric research*, vol. 61, no. 3, pp. 239–250, 2002.

[66] J. W. Choi and N. Kim, "Clinical application of three-dimensional printing technology in craniofacial plastic surgery," *Archives of plastic surgery*, vol. 42, no. 3, pp. 267–277, 2015.

[67] S. L. Sing, J. An, W. Y. Yeong, and F. E. Wiria, "Laser and electron-beam powder-bed additive manufacturing of metallic implants: A review on processes, materials and designs," *Journal of Orthopaedic Research*, 2015.

[68] S. C. Joshi and A. A. Sheikh, "3d printing in aerospace and its long-term sustainability," *Virtual and Physical Prototyping*, pp. 1–11, 2015.

[69] L. Murr, S. Quinones, S. Gaytan, M. Lopez, A. Rodela, E. Martinez, D. Hernandez, E. Martinez, F. Medina, and R. Wicker, "Microstructure and mechanical behavior of ti–6al–4v produced by rapid-layer manufacturing, for biomedical applications," *Journal of the mechanical behavior of biomedical materials*, vol. 2, no. 1, pp. 20–32, 2009.

[70] Y.-L. Pan, J. Bowersett, S. C. Hill, R. G. Pinnick, and R. K. Chang, "Nozzles for focusing aerosol particles," ARMY RESEARCH LAB ADELPHI MD COMPUTATIONAL AND INFORMATION SCIENCES DIRECTORATE, Tech. Rep., 2009.

[71] K. Moreland, B. Geveci, K.-L. Ma, and R. Maynard, "A classification of scientific visualization algorithms for massive threading," in *Proceedings of the 8th International Workshop on Ultrascale Visualization*, 2013, pp. 1–10.

[72] "Paraview." [Online]. Available: https://www.paraview.org/

[73] "tecplot." [Online]. Available: https://www.tecplot.com/

[74] "Mayavi." [Online]. Available: https://docs.enthought.com/mayavi/mayavi/overview.html

[75] "Vmd." [Online]. Available: https://www.ks.uiuc.edu/Research/vmd/

[76] M. Rivi, L. Calori, G. Muscianisi, and V. Slavnic, "In-situ visualization: State-of-the-art and some use cases," *PRACE White Paper*, pp. 1–18, 2012.

[77] "Amazon web service (aws) - cloud computing service." [Online]. Available: https://aws.amazon.com/

[78] "Google cloud." [Online]. Available: https://cloud.google.com/

[79] "Microsoft azure." [Online]. Available: https://azure.microsoft.com/sv-se/

[80] "Ibm cloud." [Online]. Available: https://www.ibm.com/cloud

[81] "Nvidia gpu in google cloud." [Online]. Available: https://cloud.google.com/nvidia/

[82] "Using clusters for large-scale technical computing in the cloud." [Online]. Available: https://cloud.google.com/solutions/using-clusters-for-large-scale-technical-computing

[83] "Introducing amazon ec2 c5n instances featuring 100 gbps of network bandwidth." [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-c5n-instances/

[84] "Kubernetes." [Online]. Available: https://kubernetes.io/

[85] "Elasticluster." [Online]. Available: https://elasticluster.readthedocs.io/en/latest/

[86] "Create a grid engine cluster on compute engine." [Online]. Available: https://googlegenomics.readthedocs.io/en/latest/use_cases/setup_gridengine_cluster_on_compute_engine/

[87] "Elasticluster to access your gcp project and generate an ssh key pair." [Online]. Available: https://cloud.google.com/genomics/docs/tutorials/grid-engine-cluster

[88] "My slurm cluster." [Online]. Available: https://cloud.google.com/solutions/running-r-at-scale

[89] "Salome." [Online]. Available: https://www.salome-platform.org/user-section/about/mesh

[90] "Gmsh." [Online]. Available: http://gmsh.info/

[91] "Pointwise, the choice for cfd meshing." [Online]. Available: https://www.pointwise.com/

[92] "Ansa-the advanced cae pre-processing software for complete model build up." [Online]. Available: https://www.beta-cae.com/ansa.htm

[93] H. P. Babbage, "Babbage analytical engine," in *The Origins of Digital Computers*. Springer, 1982, pp. 67–70.

[94] A. Huang, "Architectural considerations involved in the design of an optical digital computer," *Proceedings of the IEEE*, vol. 72, no. 7, pp. 780–786, 1984.

[95] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[96] ——, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[97] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[98] NVIDIA, *Volta Architecture*, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[99] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[100] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 73–82.

[101] J. A. Jablin, T. B. Jablin, O. Mutlu, and M. Herlihy, "Warp-aware trace scheduling for GPUs," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 163–174.

[102] Top500 supercomputing sites. [Online]. Available: https://www.top500.org/lists/2014/11/

[103] M. Sourouri, T. Gillberg, S. B. Baden, and X. Cai, "Effective multi-gpu communication using multiple cuda streams and threads," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2014, pp. 981–986.

[104] Gpudirect. [Online]. Available: https://developer.nvidia.com/gpudirect

[105] T. Gillberg, M. Sourouri, and X. Cai, "A new parallel 3d front propagation algorithm for fast simulation of geological folds," *Procedia Computer Science*, vol. 9, pp. 947–955, 2012.

[106] W.-K. Jeong and R. T. Whitaker, "A fast iterative method for eikonal equations," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2512–2534, 2008.

[107] O. Weber, Y. S. Devir, A. M. Bronstein, M. M. Bronstein, and R. Kimmel, "Parallel algorithms for approximation of distance maps on parametric surfaces," *ACM Transactions on Graphics (TOG)*, vol. 27, no. 4, p. 104, 2008.

[108] T. Gillberg, "Fast and accurate front propagation for simulation of geological folds," 2013.

[109] E. Krishnasamy, "Hybrid cpu-gpu parallel simulations of 3d front propagation," 2014.

[110] The erik gpu cluster at lunarc. [Online]. Available: URLhttp://www.lunarc.lu.se/Systems/ErikDetails

[111] An overview of zorn, pdcâĂŹs gpu cluster. [Online]. Available: URLhttps://www.pdc.kth.se/resources/computers/zorn

[112] D. A. Ibanez, "Conformal mesh adaptation on heterogeneous supercomputers," *Ph. D. thesis, Rensselaer Polytechnic Institute*, 2016.

[113] P. Valero, J. L. Sánchez, D. Cazorla, and E. Arias, "A gpu-based implementation of the mrf algorithm in itk package," *The Journal of Supercomputing*, vol. 58, no. 3, pp. 403–410, 2011.

[114] P. Valero-Lara, A. Pinelli, J. Favier, and M. P. Matias, "Block tridiagonal solvers on heterogeneous architectures," in *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 2012, pp. 609–616.

[115] P. Valero-Lara, A. Pinelli, and M. Prieto-Matias, "Fast finite difference poisson solvers on heterogeneous architectures," *Computer Physics Communications*, vol. 185, no. 4, pp. 1265–1272, 2014.

[116] P. Valero-Lara, "Accelerating solid–fluid interaction based on the immersed boundary method on multicore and gpu architectures," *The Journal of Supercomputing*, vol. 70, no. 2, pp. 799–815, 2014.

[117] P. Valero-Lara, F. D. Igual, M. Prieto-Matías, A. Pinelli, and J. Favier, "Accelerating fluid–solid simulations (lattice-boltzmann & immersed-boundary) on heterogeneous architectures," *Journal of Computational Science*, vol. 10, pp. 249–261, 2015.

[118] S. A. Kalray, "Mppa accesscore posix progamming reference manual," 2013.

[119] B. D. De Dinechin, D. Van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.

[120] P. Valero-Lara and F. L. Pelayo, "Full-overlapped concurrent kernels," in *ARCS 2015-The 28th International Conference on Architecture of Computing Systems. Proceedings*. VDE, 2015, pp. 1–8.

[121] P. Valero-Lara, P. Nookala, F. L. Pelayo, J. Jansson, S. Dimitropoulos, and I. Raicu, "Many-task computing on many-core architectures," *Scalable Computing: Practice and Experience*, vol. 17, no. 1, pp. 32–46, 2016.

[122] M. D. Gomony, B. Akesson, and K. Goossens, "Coupling tdm noc and dram controller for cost and performance optimization of real-time systems," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.

[123] H. Zhao, O. Jang, W. Ding, Y. Zhang, M. Kandemir, and M. J. Irwin, "A hybrid noc design for cache coherence optimization for chip multiprocessors," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 834–842.

[124] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Exploration of distributed shared memory architectures for noc-based multiprocessors," *Journal of Systems Architecture*, vol. 53, no. 10, pp. 719–732, 2007.

[125] D. M. Young, *Iterative solution of large linear systems*. Elsevier, 2014.

[126] B. D. de Dinechin, Y. Durand, D. Van Amstel, and A. Ghiti, "Guaranteed services of the noc of a manycore processor," in *Proceedings of the 2014 International Workshop on Network on Chip Architectures*. ACM, 2014, pp. 11–16.

[127] T. Goubier, R. Sirdey, S. Louise, and V. David, "σc: A programming model and language for embedded manycores," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2011, pp. 385–394.

[128] P. Valero-Lara and J. Jansson, "Lbm-hpc-an open-source tool for fluid simulations. case study: unified parallel c (upc-pgas)," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 318–321.